Comparing the Quality of Code Generated by Vibe Coding Tools

Gustavo da Mota
Universidade Federal de Pernambuco (UFPE)
Recife, Brazil
gpsm@cin.ufpe.br

Kiev Gama
Universidade Federal de Pernambuco (UFPE)
Recife, Brazil
kiev@cin.ufpe.br

Abstract

The use of AI agents for automatic code generation has become increasingly common in software development. However, concerns remain about the quality of the generated code, including aspects of maintainability, readability, and long-term evolution. This study compares the structural quality of code produced by three widely adopted vibe coding tools — Lovable, v0, and Replit — starting from a single generation prompt. We generate three independent projects per tool, totalling nine web applications, and submit them to static analysis with SonarQube. We collect metrics such as the number of issues, severity distribution, estimated remediation effort, cyclomatic and cognitive complexity, and code duplication. Preliminary results show that the tools exhibit distinct qualitative profiles: Lovable concentrates issues of lower severity but presents a substantially higher density of code smells per KLOC, while v0 and Replit produce more code with more aggressive severity profiles. These findings suggest that choosing between vibe coding tools involves structural trade-offs that go beyond perceived productivity.

Keywords

Vibe coding, Code smells, Static analysis, SonarQube, Code quality, AI code generation

1 Introduction

With the growing adoption of AI-based code generation tools in software development — whose usage rose from 76% in 2024 to 84% in 2025 according to the Stack Overflow Developer Survey [15] — it becomes possible to develop entire applications from a small set of natural-language instructions. This new possibility brings not only potential gains in productivity, reported by 69% of AI-agent users [15], but also concerns about the quality, readability, organization, and maintainability of the generated code [3, 12]. Empirical studies reinforce these concerns: Pearce et al. [11] found that around 40% of the programs completed by GitHub Copilot in security-relevant scenarios contained vulnerabilities, while evaluations spanning multiple tools report wide variation in the correctness, security, and maintainability of the generated code [18].

Already consolidated in software engineering for analyzing code written by human developers, static analysis can serve as an important instrument to assess code quality. As reported by Trautsch et al. [17], the accumulation of issues identified by static analysis tools tends to be associated with a higher number of real problems during software execution.

In this context, this study proposes to compare AI-based application generation tools through the static analysis of the code they produce. We seek to identify differences between the evaluated tools, as well as recurring patterns of structural problems present in the produced code.

2 Background

2.1 Vibe Coding and AI-Assisted Code Generation

The notion of vibe coding has been used to describe an emerging way of building software, in which the developer interacts in natural language with AI tools that take over most of the structural decisions of the system. The term was popularized by Andrej Karpathy in 2025, when he described a practice in which the developer *"gives in"* to the flow of interaction with generative models and steers the system more by intent than by the literal writing of code [6]. Recent surveys highlight the importance of evaluating other aspects of the code (e.g., security, maintenance, readability, and adherence to good practices), and point to a dichotomy between speed and quality, with users reporting a considerable trade-off between speed of *"functionality"* and structural and maintenance inconsistencies [3, 5]. Unlike traditional programming assistants, whose main focus is to help writing or editing code, platforms such as Lovable, v0, and Replit AI propose development flows with a high level of abstraction and automated application generation [3]. This widens the research gap in software engineering: instead of evaluating isolated functions or algorithmic solutions, it became necessary to examine the structural quality of entire AI-generated applications.

2.2 Code Quality, Code Smells, and Tech Debt

The notion of code quality involves multiple dimensions, and different analysis tools vary in the relevance they assign to each evaluation aspect. Models such as ISO/IEC 25010 separate these aspects into categories such as maintainability, reliability, security, performance efficiency, and portability.

Fowler et al. [4] introduced the concept of code smells, which comprises surface code patterns that may indicate deeper problems of maintenance, comprehensibility, and defects in the system. The relevance of code smells stems from the fact that they do not necessarily cause immediate failures, but can increase complexity, hinder maintenance, and contribute to the accumulation of technical debt. Tech debt can in turn be understood as the future cost — in time and effort — associated with decisions that facilitate short-term delivery but produce instability and difficulty of future maintenance. Static analysis tools such as SonarQube aim to detect part of these concepts through automated metrics and rules. SonarQube classifies problems into categories such as bugs, vulnerabilities, and code smells, defining a code smell as a maintainability problem that makes the code confusing and hard to maintain. The tool also computes metrics such as technical debt and technical debt ratio, associating an estimated remediation effort with the issues found. As shown by Trautsch et al. [17], the accumulation of issues reported by static analysis usually indicates a higher number of real problems in the system.

Thus, using static analysis tools to detect code smells in applications generated by vibe coding tools, while not replacing functional testing and manual review, represents a first step to investigate the structural quality of the code.

3 Methodology

Rather than declaring one tool superior to the others, the goal of this study is to obtain initial evidence about the structural quality of code generated by vibe coding tools from the same requirements specification. Adopting an exploratory comparative design, suitable for the Innovative Ideas and Emerging Results track, we aim to answer the following research question:

RQ1: *How do vibe coding tools differ in the number and severity of code smells detected in web applications generated from the same set of requirements?*

The unit of analysis is a project generated by a web application generation platform, fulfilling the functional requirements stated in a prompt that is identical for all interactions with the models.

3.1 Prompt Design

The prompt was designed to elicit applications with a minimum level of complexity, while not providing the model with examples of previous implementations to draw upon. This approach, known as zero-shot [2], is characterized by the absence of code or prior implementations in the prompt, so that the model produces a response based exclusively on the supplied instructions.

The requirements were based on an example application called *"Nature Park Wildlife App"* [1], in which users can register wildlife sightings, report safety issues, and subscribe to notifications about specific animals, supported by a geolocated map. The full prompt is included in the research artifacts.

3.2 Tools Under Evaluation

The selected tools were Lovable, v0, and Replit. They are three widely used platforms, completely web-based, that aim to generate entire applications from a few natural-language interactions and that offer free tiers with token quotas sufficient to support the experiments.

As these are continuously updated SaaS platforms, the tools do not expose public version numbers; we therefore report the generation window of the projects — between May 6 and May 21, 2026 — together with the build identifiers observable in the artifacts. On Lovable, the projects were generated from the `tanstack_start_ts` template (dated 2026-05-06 and 2026-05-12) by the `gpt-engineer-app` agent; on v0, by the *v0.app* platform, producing Next.js 16.2.4 applications; and on Replit, by the *Replit Agent* (Nix channel `stable-25_05`, Node.js 24).

3.3 Generation Procedure

Each tool was exposed to the same prompt, with no additional instructions or manual corrections. The outcome of each iteration was documented and the projects were stored in separate repositories. The process was repeated three times per tool, in order to minimize the influence of atypical outputs on the final results, which are possible due to the non-deterministic nature of the models.

This procedure yielded nine distinct projects:

- **Lovable:**
  `wildlife-whisperer-app`,
  `wild-park-patrol`,
  `nature-watch-live`.
- **v0:**
  `v0-aplicativo-de-rastreamento`,
  `v0-wildlife-tracking-app`,
  `v0-wildlife-track-app`.
- **Replit:**
  `Wildlife-Tracker-replit-1`,
  `Wildlife-Tracker-replit-2`,
  `Nature-Watch-System-replit-3`.

3.4 Static Analysis Procedure

The evaluation was independent for each project and used SonarQube as the static analysis tool. SonarQube is among the most widely adopted static analysis tools in both industry and software engineering research [1, 7], which favors the comparability and reproducibility of our results. SonarQube produces a list of issues found in the project. Each issue contains: violated rule, severity, source component, line, message, type, tags, and remediation effort. In addition, project-level information is produced, such as duplicated-line density, cyclomatic complexity, cognitive complexity, and non-commented lines of code [14].

3.4.1 Rules, Tags, and Issue Types. The rule is more literal regarding which basic rule was violated for that issue to be raised, while tags are broader signifiers that give meaning to the problem itself and can be more than one per issue. For instance, the use of a deprecated API will trigger rule S1874, and the tags of that issue will include the category *obsolete*. Issue types fall into three main categories: code smells, bugs, and vulnerabilities.

3.4.2 Remediation Effort. The effort metric represents an estimated time to remediate the issue, providing a possible measure of technical debt. From the available data, we can not only compare the tools numerically, but also compare qualitatively the type of problem each tool is more prone to generate, as well as their severity and remediation effort.

3.5 Comparison Strategy

For comparison, we primarily use the number of issues per project, the associated effort, and the proportional distribution of their types and tags.

3.5.1 Quantitative Comparison. The quantitative comparison has two axes: raw and proportional. Each tool produced a different volume of code, which makes a proportional analysis necessary to avoid misleading conclusions from raw numbers. On the other hand, the expected functionality of the projects is the same, which makes the non-proportional analysis also relevant, as it is, in a sense, proportional to the delivered functionality.

3.5.2 Qualitative Comparison. The qualitative comparison examines, within each project, the predominant patterns of each tool.

[1] This activity was originally developed for Michel Chaudron's classes at Eindhoven University of Technology and also used in a hands-on session of the 3rd Designing Workshop at ICSE 2026

To this end, we primarily rely on the severity distribution of issues and, as a complementary indicator, on their tags, chosen for their descriptive nature. The tags generated by the issues of the nine projects amount to thirty-nine distinct categories, which were grouped into five clusters to improve data visualization: Semantic, Technical Quality, Security, Technological Context, and Language / Ecosystem. The complete mapping of tags into these clusters is presented in Table 1.

Table 1: Mapping of tags into clusters.

Cluster	Tags
Semantic	readability, brain-overload, confusing, redundant, suspicious, pitfall, consistency, unused, error-handling, bad-practice
Technical Quality	performance, accessibility, optimization
Security	cwe
Technological Context	react, jsx, html, javascript, regex, async, import, esm-only, eslint, wcag, nodejs, linting
Language / Ecosystem	es2015, es2020, es2021, es2022, internationalization, portability, formatting, convention, type-dependent, obsolete, nullish-coalescing, modernize, unicode

4 Data Presentation

This section presents the data extracted from the static analysis of the nine generated projects. For each evaluated tool, we detail the per-project information (and the aggregated total), followed by the profiles of issue types, severity, tags, and rules. Section 4.4 then consolidates the data into a direct comparison between the three tools, in raw values and normalized per KLOC.

Before the per-tool breakdown, Table 2 describes the SonarQube rules that appear in the rule-frequency charts of the three tools (Figures 2, 4, and 6), so that the subsequent analysis can refer to them directly. The descriptions follow the official SonarQube rule documentation [13].

Table 2: SonarQube most recurring rules appearing in the per-tool rule-frequency charts (`typescript:` prefix omitted).

Rule	Description
S1874	Deprecated APIs, classes, or functions should not be used.
S3358	Ternary operators should not be nested.
S4325	Redundant casts and non-null assertions should be removed.
S6481	React Context Provider values should have stable identities.
S6759	React component props should be read-only.
S6819	Prefer a semantic HTML tag over an ARIA *role*.
S7735	Negated conditions should be avoided when an *else* clause is present.
S7748	Number literals should not have unnecessary decimal points or trailing zeros.
S7764	Use `globalThis` instead of `window`, `self`, or `global`.
S7773	`Number` static methods and properties should be preferred over their global equivalents.
S7781	Use `replaceAll()` instead of `replace()` with a global *regex*.

4.1 Lovable

Table 3 summarizes the raw metrics for the three projects generated by Lovable. The tool produced the lowest total volume of code among the three evaluated tools and introduced only a single bug according to the SonarQube classification, but it concentrated the highest density of code smells per line of code.

Table 3: Per-project metrics — Lovable.

Project	NCLOC	Issues	Eff.(min)	Cog.Cplx
wildlife-whisperer-app	5,781	59	215	225
wild-park-patrol	5,697	167	1,782	237
nature-watch-live	5,603	164	1,774	219
Total	**17,081**	**390**	**3,771**	**681**

As shown in the table, the `wildlife-whisperer-app` project yielded considerably different results from the other two projects, at least in the issues and effort metrics. This divergence is plausible and consistent with the discussion of model non-determinism, and reinforces the importance of multiple iterations per tool to minimize the impact of anomalous outputs.

A characteristic of Lovable is that the vast majority of its issues were classified as *Code Smell* (389 out of 390), in contrast to the other tools, which presented more varied type profiles. In addition, the severity of the detected problems was mostly MINOR, as can be seen in Figure 1, with `wildlife-whisperer-app` once again showing a large gap with respect to the other projects of the tool.

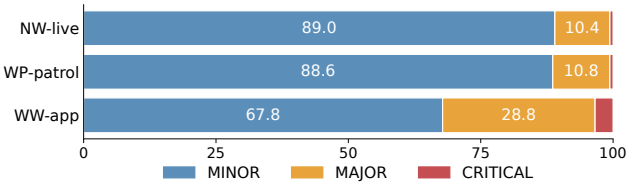

Figure 1: Severity distribution per project — Lovable (%).

The most striking aspect of Lovable's data is the predominance of issues raised by rule S1874, which targets obsolete APIs and deprecated classes and functions (Table 2). As Figure 2 shows, this signature will also be visible in the cross-tool comparison.

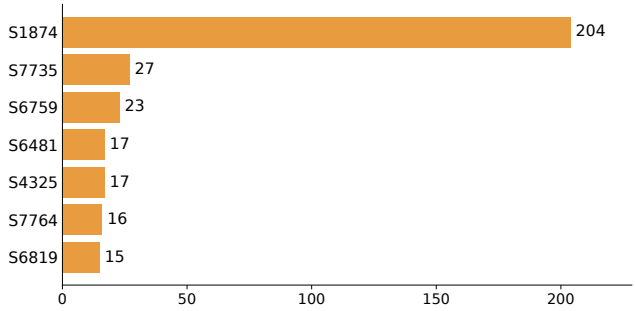

Figure 2: Most frequent rules in Lovable's issues (top 7, `typescript:` prefix omitted).

4.2 v0

Table 4 summarizes the raw metrics for the three projects generated by v0. The tool produced an intermediate code volume and the lowest total remediation effort, with only a few issues classified as bugs.

Table 4: Per-project metrics — v0.

Project	NCLOC	Issues	Eff.(min)	Cog.Cplx
v0-aplicativo-de-rastreamento	8,790	77	284	277
v0-wildlife-tracking-app	9,278	85	283	305
v0-wildlife-track-app	9,515	92	374	354
Total	**27,583**	**254**	**941**	**936**

v0 positioned itself as intermediate on several metrics, which is especially visible in the lines-of-code count: all three projects stayed close to nine thousand lines, with low standard deviation across projects. Regarding severity (Figure 3), it also remained intermediate in the overall proportion of severity levels.

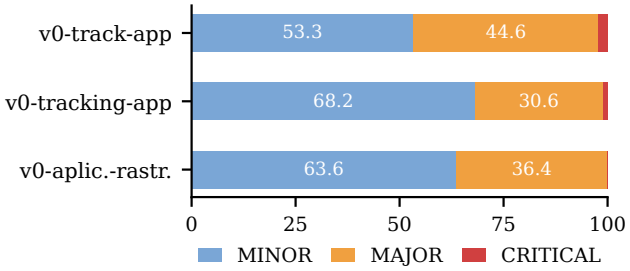

Figure 3: Severity distribution per project — v0 (%).

Regarding the rules most violated by v0's code, the most salient feature is the more distributed profile: no single problem dominates as in Lovable. As for the nature of the issues, they relate more to framework-specific best practices, in addition to rules `S7735` and `S7748`, which capture readability and numeric-literal formatting problems (Figure 4; Table 2).

4.3 Replit

Table 5 presents the raw metrics for the three projects generated by Replit. The tool produced the largest total volume of code, with substantial line duplication and the broadest coverage of distinct rules (35).

Table 5: Per-project metrics — Replit.

Project	NCLOC	Issues	Eff.(min)	Cog.Cplx
Wildlife-Tracker-replit-1	16,827	177	632	655
Wildlife-Tracker-replit-2	15,736	140	520	558
Nature-Watch-System-replit-3	15,848	146	536	602
Total	**48,411**	**463**	**1,688**	**1,815**

At first glance, the high number of lines of code produced by Replit stands out. This is also reflected in the number of duplicated lines, with an average of about 60% — a remarkably high

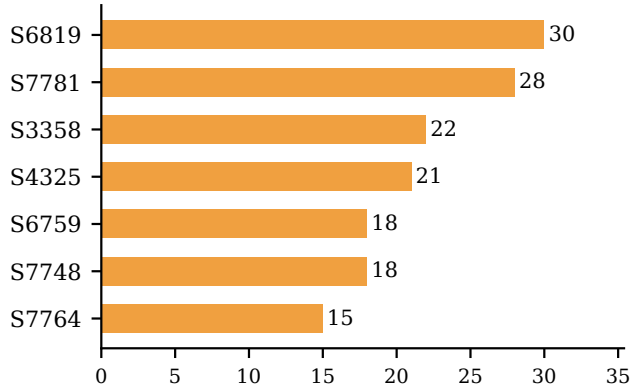

Figure 4: Most frequent rules in v0's issues (top 7, `typescript:` prefix omitted).

figure considering that the next tool with most duplication, v0, never exceeded 7%. Another negative metric was the severity of the generated problems: Replit produced, among the three tools, the highest absolute and relative share of severe issues (Figure 5).

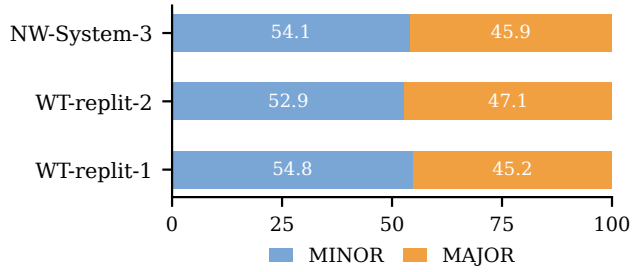

Figure 5: Severity distribution per project — Replit (%).

Regarding the rules violated in Replit's projects, two stand out (Figure 6): `S4325`, related to redundant type assertions, and `S6819`, an accessibility rule that recommends semantic HTML elements instead of ARIA *role* attributes (Table 2).

4.4 Tool Comparison

For the cross-tool comparison, it is important to present the data both in absolute form and in relative form — in this case, per thousand lines of code. Raw figures may seem to favor tools that produce less code (and thus less surface for errors), but they remain essential: there is no merit in fulfilling the same functional requirements with more lines of code, especially when there is massive duplication, as in the case of Replit.

Table 6 consolidates, per tool, the raw totals and the same values normalized per KLOC (thousand non-commented lines of code). The effort metric is expressed in estimated remediation minutes.

4.4.1 Remediation Effort. Figure 7 contrasts the total number of issues and the total remediation effort per tool, in absolute terms. Although Lovable has fewer issues than Replit, it has the largest

Table 6: Comparison across tools: raw totals and KLOC-normalized values.

Tool	NCLOC	Issues	CS	Bugs	Hot.	Eff.(min)	D.Rules
Lovable	17,081	390	389	1	6	3,771	24
v0	27,583	254	249	5	10	941	24
Replit	48,411	463	455	8	30	1,688	35
Normalized per KLOC							
Lovable	—	22.83	22.77	0.06	0.35	220.77	—
v0	—	9.21	9.03	0.18	0.36	34.12	—
Replit	—	9.56	9.40	0.17	0.62	34.87	—

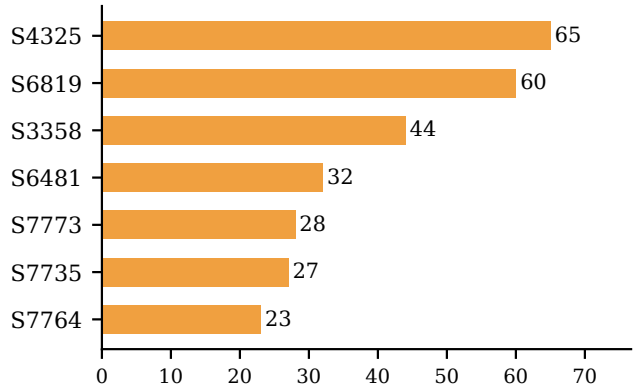

Figure 6: Most frequent rules in Replit's issues (top 7, `typescript:` prefix omitted).

remediation-time requirement, showing that the lower severity of Lovable's issues does not translate into low correction effort.

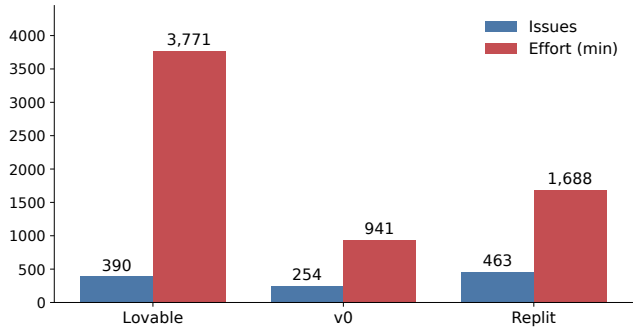

Figure 7: Absolute comparison: total number of issues and total remediation effort (in minutes) per tool.

4.4.2 Qualitative Profile of Issues. Figure 8 shows the tag profile aggregated by *cluster*, in proportion (%), allowing a qualitative comparison of which structural dimensions each tool concentrates its problems in. Lovable stands out for the heaviest concentration on the security cluster and, above all, on the language / ecosystem cluster — the latter strongly driven by rule S1874. Replit and v0 show qualitative profiles similar to each other.

As for the severity distribution among the three tools in percentages (Figure 9), the shift from Lovable's profile (predominantly

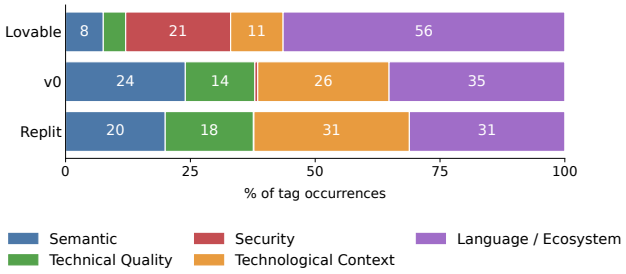

Figure 8: Tag profile by *cluster* (% per tool).

MINOR) to Replit's profile (nearly balanced between MINOR and MAJOR) is evident, with Replit being the only tool not producing any CRITICAL-classified problem. Lovable and v0 produce small shares of CRITICAL issues (1.0% and 1.2% respectively).

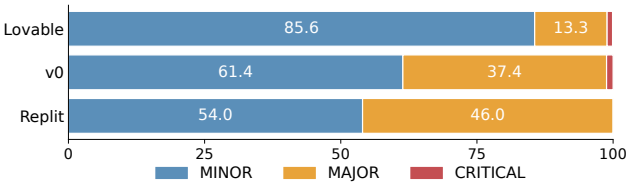

Figure 9: Severity distribution per tool (percentages).

5 Discussion

The findings of this study highlight a trade-off that is practically invisible in the adoption of vibe coding tools: reported productivity gains do not necessarily reflect structural quality. The Lovable case illustrates this paradox — the tool presented an intermediate issue count in static analysis, yet the highest remediation effort among the evaluated tools, suggesting that surface-level metrics may induce a false perception of quality.

The predominance of rule S1874 in the projects generated by Lovable — deprecated APIs accounting for 204 occurrences — finds an explanation in the literature on LLMs for code generation. Models trained on large historical corpora tend to over-represent API patterns that were prevalent at the time of training, even when those patterns have since been formally discontinued. This phenomenon, documented by Lin et al. [8] and by subsequent studies on knowledge conflicts in LLMs, suggests that vibe coding tools

may systematically generate code oriented toward the past of the ecosystem — not toward its current state.

This risk takes on an additional dimension when the target audience of these platforms is considered. Tools such as Lovable are explicitly positioned for non-developer users, lowering the barrier to creating complete web applications. However, this same accessibility eliminates the technical review layer that, in traditional development contexts, would serve as a filter for structural and security issues. Documented incidents — such as `CVE-2025-48757`, which exposed user data in more than 170 applications generated by the platform [10, 16] — illustrate the concrete consequences of this gap. The present study contributes precisely by making this trade-off measurable: not as a criticism of the tools themselves, but as evidence that their adoption requires awareness of the structural debt profiles that each tends to produce.

6 Limitations

The main methodological challenge of this study refers to the capacity of static analysis to sufficiently quantify the code quality of a software project. Recent literature positions SonarQube as a relevant auxiliary indicator, but not as an instrument capable of definitively assessing code quality. Marcilio et al. [9], in a study spanning 421,976 issues from 246 projects across four distinct SonarQube instances, observed that on average only 13% of the issues reported by the tool are effectively fixed — and conjecture that only a subset of its checkers actually reveal real design and coding flaws, which may artificially inflate the measured technical-debt index.

A second limitation concerns the sampling scope of the study. Due to token-credit restrictions in the free versions of the tools and the project's time horizon, only three projects were generated per tool — a volume still susceptible to anomalous outputs. The collected data illustrate this risk concretely: in the set of projects generated by Lovable, the standard deviation of the code-smells metric was 62.1, considerably higher than that observed for v0 (SD = 7.0) and Replit (SD = 20.3). This dispersion is explained, to a large extent, by the project `wildlife-whisperer-app`, which recorded only 58 code smells — a substantial divergence from the other projects of the same tool (164 and 167, respectively) — and which, on its own, disproportionately distorts the tool's mean. With larger samples, such fluctuations would tend to be diluted, making the comparisons between tools more robust and less sensitive to point variations of a single execution.

Finally, a third limitation concerns the opacity of the platforms under evaluation: there is no public transparency about how each platform allocates computational resources among its users in the free tiers. It is plausible to speculate — and this is speculation, not a claim — that, at peak times, lower-capacity models may be invoked in place of the platform's default model, affecting the quality of the generated code. This factor reinforces the need for broader replications across varied contexts before drawing definitive conclusions.

7 Conclusion and Future Work

This study aimed to provide initial evidence that can guide the choice between vibe coding tools from the perspective of the structural quality of the generated code. Taken together, the data suggest that each of the three evaluated tools presents its own risk profile, with distinct implications for adopting them in a real development workflow. Among the three, v0 emerged as the most balanced alternative. It produced the lowest total remediation effort (941 minutes, against 3,771 for Lovable and 1,688 for Replit), and intermediate severity proportions, with neither the peak of MAJOR issues observed in Replit nor the strong concentration on deprecated APIs observed in Lovable.

Lovable exposes a relevant trade-off: although the severity of its issues is largely MINOR (∼86%), the tool concentrated the highest estimated remediation effort (∼4.0× v0's), with more than half of that effort tied to a single rule (`S1874`, on deprecated APIs and functions; Table 2). This specific concentration, combined with the weight of the *cwe* tag on the security cluster, suggests that Lovable may produce code that looks clean but relies on APIs whose reliability — including in security terms — has already been questioned by the ecosystem itself. It should be noted, as discussed in Section 6, that part of Lovable's average is pulled by the `wildlife-whisperer-app` project, whose divergent behavior relative to the other Lovable projects calls for caution when interpreting these averages in isolation.

Replit, in turn, presents two characteristics that deserve special attention. The first is the massive line duplication, with an average of approximately 60% across all three projects — a figure substantially above v0 (≤7%) and Lovable (0%), indicating strong structural verbosity. The second is the highest absolute and proportional share of MAJOR-severity problems (213 occurrences, 46% of the total), above the other two tools. Combined, these two factors suggest that the productivity gain from using Replit may come with a more significant structural debt over the medium term.

As future work, we intend to: (i) significantly increase the number of iterations per tool, mitigating the effect of anomalous outputs such as the one observed in `wildlife-whisperer-app`; (ii) submit the *premium* versions of these platforms to the same procedure, under the hypothesis that differences in model tier may materially affect the results; and (iii) incorporate other static analysis tools besides SonarQube, in order to reduce the bias of single-tool coverage and make the comparative reading more robust.

Artifact Availability

The research artifacts — complete prompt used in every generation, source code of the nine generated projects and full SonarQube reports — are available at https://github.com/gustavo-capyba/comparing-vibe-coding-tools-artifacts.

Acknowledgements

Claude was used to generate all the figures and tables in this paper, based on explicit specifications and guidance from the authors. Additionally, Claude assisted in editing and improving the overall quality of the text, contributing to clarity, grammar, and structure. For partially supporting this work, we would like to thank INES.IA (National Institute of Science and Technology for Software Engineering Based on and for Artificial Intelligence) www.ines.org.br, CNPq grant 408817/2024-0.

References

[1] Paris C. Avgeriou, Davide Taibi, Apostolos Ampatzoglou, Francesca Arcelli Fontana, Terese Besker, Alexander Chatzigeorgiou, Valentina Lenarduzzi, Antonio Martini, Athanasia Moschou, Ilaria Pigazzini, Nyyti Saarimäki, Darius Sas, Saulo Soares de Toledo, and Angeliki-Agathi Tsintzira. 2021. An Overview and Comparison of Technical Debt Measurement Tools. *IEEE Software* 38, 3 (2021), 61–71. doi:10.1109/MS.2020.3024958

[2] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D. Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems (NeurIPS 2020, Vol. 33)*. Curran Associates, Inc., 1877–1901. https://arxiv.org/abs/2005.14165.

[3] Ahmed Fawzy, Amjed Tahir, and Kelly Blincoe. 2025. Vibe Coding in Practice: Motivations, Challenges, and a Future Outlook – A Grey Literature Review. arXiv:2510.00328 [cs.SE] https://arxiv.org/abs/2510.00328 arXiv preprint.

[4] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. 1999. *Refactoring: Improving the Design of Existing Code* (1 ed.). Addison-Wesley Professional, Boston, MA.

[5] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A Survey on Large Language Models for Code Generation. arXiv:2406.00515 [cs.SE] https://arxiv.org/abs/2406.00515 arXiv preprint.

[6] Andrej Karpathy. 2025. There's a New Kind of Coding I Call "Vibe Coding". Post on X (formerly Twitter), https://x.com/karpathy/status/1886192184808149383. Accessed 2026-05.

[7] Valentina Lenarduzzi, Fabiano Pecorelli, Nyyti Saarimäki, Savanna Lujan, and Fabio Palomba. 2023. A Critical Comparison on Six Static Analysis Tools: Detection, Agreement, and Precision. *Journal of Systems and Software* 198 (2023), 111575. doi:10.1016/j.jss.2022.111575

[8] Guancheng Lin, Xiao Yu, Jacky Keung, Xing Hu, Xin Xia, and Alex X. Liu. 2025. Lightweight Model Editing for LLMs to Correct Deprecated API Recommendations. arXiv:2511.21022 [cs.SE] https://arxiv.org/abs/2511.21022 arXiv preprint.

[9] 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 *Proceedings of the 27th International Conference on Program Comprehension (ICPC '19)*. IEEE Press, 209–219. doi:10.1109/ICPC.2019.00040

[10] MITRE Corporation. 2025. CVE-2025-48757. https://www.cve.org/CVERecord?id=CVE-2025-48757. Accessed 2026-05.

[11] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*. 754–768. doi:10.1109/SP46214.2022.9833571

[12] Mohammed Latif Siddiq, Shafayat H. Majumder, Maisha R. Mim, Sourov Jajodia, and Joanna C. S. Santos. 2022. An Empirical Study of Code Smells in Transformer-based Code Generation Techniques. In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 71–82. doi:10.1109/SCAM55253.2022.00014

[13] SonarSource. 2025. SonarQube: JavaScript/TypeScript Static Analysis Rules. https://rules.sonarsource.com/typescript/. Accessed 2026-05.

[14] SonarSource. 2025. SonarQube Server Documentation — Issues. https://docs.sonarsource.com/sonarqube-server/latest/user-guide/issues/. Accessed 2026-05.

[15] Stack Overflow. 2025. Stack Overflow Developer Survey 2025. https://survey.stackoverflow.co/2025/. Accessed 2026-05.

[16] Superblocks. 2025. Lovable Vulnerabilities: A Security Analysis. https://www.superblocks.com/blog/lovable-vulnerabilities. Accessed 2026-05.

[17] Alexander Trautsch, Steffen Herbold, and Jens Grabowski. 2023. Are Automated Static Analysis Tools Worth It? An Investigation into Relative Warning Density and External Software Quality on the Example of Apache Open Source Projects. *Empirical Software Engineering* 28, 3 (2023), 66. doi:10.1007/s10664-023-10301-2

[18] Burak Yetiştiren, Işık Özsoy, Miray Ayerdem, and Eray Tüzün. 2023. Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT. arXiv:2304.10778 [cs.SE] https://arxiv.org/abs/2304.10778 arXiv preprint.