

The Adoption of Vibe Coding and the Transformation of Programming Practices Induced by Generative Models: A SonarQube-Based Analysis

Maja Benkus Kumek, Neven Vrčak

University of Zagreb Faculty of Organization and Informatics

Pavlinka ul. 2, 42000 Varaždin, Croatia

{mbenkus, neven.vrcak}@foi.unizg.hr

Abstract. *This paper studies how vibe coding – specifying behaviour in natural language for generative models to implement – affects code quality. It combines a review of mostly qualitative research with a preliminary, exploratory case study using SonarQube static analysis. Three implementations of the same feature are compared: one manual and two vibe-coded with GPT-5.2-Codex and Gemini 3.1 Pro. The aim is to demonstrate a metric-based comparison and generate hypotheses, not to confirm them. Both vibe-coded versions showed fewer code smells, lower complexity, and less technical debt than the manual baseline, while a runtime defect missed by static metrics reveals the approach’s limits.*

Keywords. vibe coding, programming, software engineering, artificial intelligence, large language models, generative models, technical debt, SonarQube

1 Introduction

Large language models (LLMs) are a type of deep learning model trained on large amounts of text data, which allows them to understand and generate natural language across a wide range of tasks. Their versatility has made them one of the most impactful technologies in recent years. In addition to common uses such as artificial intelligence (AI) assistants and text processing, LLMs are increasingly being used to generate and review source code, as well as to detect bugs (Hou et al., 2024; Jiang et al., 2025).

As generative AI becomes more widespread in software development, a new paradigm has emerged known as *vibe coding*. In vibe coding, the developer describes the desired behaviour of a software product in natural language, and a generative model produces or refines the corresponding code (Sapkota et al., 2025; Sarkar and Drosos, 2025). While this approach can make software development faster and more accessible, it also raises concerns about code quality, readability, and long-term maintainability. These concerns motivate a closer scientific examination of how gener-

ative models influence programming practices and the quality of the code they produce.

This paper investigates the impact of vibe coding on programming practices and code quality. It combines a review of existing literature on generative models in software development with an empirical analysis of code produced using the vibe coding approach. The primary objective is to compare code quality between vibe coding and conventional development, using static code quality metrics from SonarQube as the basis for comparison. A secondary objective is to assess how well the empirical findings align with the claims made in existing research.

In accordance with the stated research objectives, the following research questions are addressed:

- **RQ1:** What research methodologies and changes in developer work practices are identified in the existing scientific literature on the application of generative models in programming?
- **RQ2:** How does the existing literature address the impact of generative models and vibe coding on code quality and technical debt?
- **RQ3:** Are there measurable differences in static code quality metrics between manually developed and vibe coding implementations, and what advantages and limitations of the vibe coding approach can be identified through SonarQube analysis?

The empirical component reported here is deliberately framed as a preliminary, exploratory case study rather than a controlled experiment; its contribution lies in establishing and demonstrating a measurable, metric-based comparison approach that future work can scale to multiple developers, tasks, and projects.

2 Related Work

The literature review covers three aspects of generative language model use in programming: the experience of developers working with these tools, changes in the software development process, and the potential effects

on code quality and long-term maintainability. Particular attention is given to studies that view vibe coding not just as a tool for boosting productivity, but as a broader shift in how responsibility is divided between human developers and AI-based systems.

2.1 Developer Role, Comprehension, and Maintainability

Research on generative tools in programming extends beyond the speed and quality of produced code to the developer's changing role and, increasingly, to the downstream consequences for code comprehension and maintenance. A body of work grounded in developer-experience surveys and self-reported assessments (Brandebusemeyer, 2025; Sampson, 2025) examines the subjective side of working with generative AI – cognitive load, control, and the pressure to adopt new tools. While these well-being concerns lie outside the scope of the present study, the same studies consistently observe that working with generated code introduces additional verification and correction effort. Weisz et al. (2025) report this pattern directly in an industrial deployment of an AI coding assistant: perceived productivity rises, but the gains are uneven across developers and are partly offset by the extra steps required to check and fix generated output, so an initial impression of higher productivity may not be sustained.

Sankaranarayanan (2026) conducted a controlled experiment with 78 participants to examine whether unrestricted AI use leads to a decline in developer competence. The study compared three groups: developers working manually, those using AI without restrictions, and those using AI with a structured scaffolding mechanism. While both AI groups performed similarly on the initial coding task, the unrestricted AI group had a 77% failure rate on a subsequent maintenance task completed without AI assistance, compared to 39% in the scaffolded group. The author attributes this gap to the accumulation of *epistemic debt* – a situation where developers own code functionally but do not understand it well enough to maintain it independently. This finding is relevant to the present study because it provides behavioural evidence for the same phenomenon that SonarQube-based static analysis aims to capture at the codebase level: that AI-generated code may appear functional while concealing underlying quality and maintainability problems.

Maes (2026) provides a broad empirical synthesis of AI-assisted software development practices in early 2026, drawing on data from developer surveys, code complexity analyses, and production environment observations. The study introduces the concept of the *Great Toil Shift*, describing how the time saved during AI-assisted code generation tends to be entirely consumed by the increased effort required for code review, security auditing, and maintenance. Quantita-

tive findings indicate that AI-assisted pull requests contained 1.7 times more structural and logical issues than those authored by humans, while cognitive complexity in repositories with heavy AI use increased by 39%. These results directly support the motivation for the empirical component of the present study and offer an external reference point for interpreting the SonarQube metrics reported here.

2.2 Vibe Coding as a New Development Practice

Several studies have sought to characterise vibe coding as an emerging development practice and to examine methodologies for its investigation. Chou et al. (2025) apply Straussian Grounded Theory analysis to twenty YouTube videos dedicated to vibe coding, identifying recurring behavioural patterns, prompt formulation strategies, and error resolution approaches in AI-mediated software development. Their contribution lies primarily in conceptualising vibe coding as a new development paradigm and in describing how generative models affect the cognitive and Organisational dimensions of programming.

A methodologically more elaborate approach is adopted by Sarkar and Drosos (2025), who reconstruct complete development workflows from multi-hour recordings. Their results demonstrate that the use of generative models does not eliminate the need for programming expertise; rather, it transforms the way that expertise is exercised. Emphasis shifts increasingly from writing code toward evaluating the quality of generated solutions, managing task context, and deciding when to rely on AI and when to intervene manually. These findings support the understanding of vibe coding as a qualitative paradigm shift rather than mere automation of programming – and it is precisely this perspective that motivates the research questions of the present study.

Sapkota et al. (2025) examine the distinction between vibe coding and agentic coding, arguing that both approaches alter the structure of developer work by displacing focus from implementation details toward supervision, evaluation, and high-level decision-making. While both can reduce routine cognitive effort and accelerate development, the authors warn that the increased need for continuous evaluation and management of unpredictable model behaviour may simultaneously lead to cognitive fatigue.

Horvat (2026) investigates how the structure of prompting affects the quality of code generated through vibe coding, comparing zero-shot prompting with prompt chaining in a web front-end development task using GPT-4o and Gemini 2.5 Pro. The results show that prompt chaining – where the task is broken into sequential steps, each building on the previous output – consistently produces code with better architectural consistency, cleaner separation of concerns, and fewer

layout errors compared to zero-shot prompting, which submits all requirements in a single prompt. Zero-shot prompting was faster but resulted in lower maintainability and less adherence to design specifications. This study is relevant to the present work as it demonstrates that the quality of AI-generated code is not only a property of the model used, but also of how the developer interacts with it – a dimension that static analysis tools such as SonarQube can help quantify across different development workflows.

2.3 Technical Debt and Code Quality

One of the first detailed studies directly addressing the relationship between vibe coding and technical debt is the work of Waseem et al. (2025), who analyse how the use of generative language models affects long-term software sustainability. The study employs an experiential and qualitative methodology based on the authors' own development projects and observations from industrial practice. A key finding is that rapid iterations and the sensation of fast progress in early development phases tend to conceal the accumulation of architectural, security, testing, and maintenance problems – a dynamic the authors describe as the tension between development flow and technical debt. The authors propose guidelines for more sustainable vibe coding practice; however, the absence of standardised quantitative code quality metrics leaves room for empirical investigations such as the present study.

Van Hurne (2025) approaches technical debt in the context of generative AI more systematically, proposing a framework aimed at preventing its accumulation in early development phases. The work connects existing theoretical frameworks of technical debt with the ways in which generative models are typically employed in software development, and emphasises the need for clear development guidelines, early identification of quality requirements, and continuous maintainability assessment. While the study does not include empirical code quality measurement via tools such as SonarQube, it explicitly highlights the need for measurable technical debt indicators – a gap that the present study directly addresses through comparative static analysis.

Casserini et al. (2026) introduce the concept of *agentic entropy* to describe the process by which autonomous coding agents gradually drift away from the intended architectural design of a software system. Because agents typically operate within a limited context window, they tend to optimise for local correctness without maintaining awareness of the broader system structure, which leads to the accumulation of what the authors call *agentic technical debt* – structural misalignments, duplicated logic, and fragile refactoring that compound over time. The authors propose a process-oriented explainability framework built around architectural seeding, reasoning monitor-

ing, and a causal reasoning graph to make agent behaviour more transparent to human reviewers. While the proposed framework addresses the problem at the level of agent oversight, the present study approaches the same concern through its measurable codebase consequences, using SonarQube to detect the structural degradation that agentic entropy produces.

Gandhi (2026) presents a systematic survey of security and quality risks associated with vibe coding and autonomous development workflows, synthesising findings from empirical studies and vulnerability reports published between 2024 and 2025. Among the key quantitative findings cited in the survey is a CodeRabbit analysis of 470 GitHub pull requests, which found that AI-generated code contains 1.7 times more defects than human-authored code overall, with security issues up to 2.74 times higher and readability issues more than three times higher. The survey also highlights the *comprehension gap* as a central mechanism behind these quality problems, arguing that developers who rely on AI without reviewing the generated code lose the ability to identify structural flaws. These findings are consistent with the theoretical motivation of the present study and provide a broader empirical context for the SonarQube-based results reported here.

Taken together, the reviewed works reveal a notable absence of studies with a strong empirical component focused on measuring technical debt and code quality through quantitative metrics. Although numerous authors point to the potential accumulation of technical debt and changes in development practices associated with vibe coding, few studies have tested these claims through measurable indicators. The empirical component of the present study directly addresses this gap.

3 Methodology

The research is based on a comparison of source code developed through three distinct implementation approaches for the same functionality within the same application. To ensure comparability of results, all three scenarios share the same code repository, the same functional requirements, the same development environment, and identical evaluation criteria.

The *BlazorWebAppMovies* repository, part of Microsoft's official Blazor samples on GitHub, serves as a reference application for learning and demonstrating web application development with Blazor and Entity Framework (EF) Core ("BlazorWebAppMovies Sample Application", 2024). Since the objective of the research was to examine the impact of vibe coding, the only differentiating factor across implementations was the development approach – manual or GitHub Copilot with two distinct generative models. GitHub Copilot provides developer support through automatic code suggestion and generation based on context and source code comments. It is founded on generative language models and is designed to increase developer produc-

tivity, while responsibility for the correctness and quality of generated code remains with the user (“GitHub Copilot Documentation”, 2024). For static code quality analysis, SonarQube Community Edition was employed, enabling static source code analysis and calculation of metrics such as bugs, vulnerabilities, code smells, and technical debt (“SonarQube Community Edition”, 2024).

3.1 Implementation Scenarios

The study encompasses three implementation scenarios realised through separate Git branches: `implementation/without_vibe_coding`, `implementation/vibe_coding_gpt52codex`, and `implementation/vibe_coding_gemini31pro`. Three separate SonarQube projects were created due to the limitations of the Community Edition, which does not support cross-branch comparison within a single project. Table 1 provides an overview of the three scenarios.

Table 1. Implementation scenario overview

Scenario	Approach	Model Used
Scenario 1	Manual development	—
Scenario 2	Vibe Coding	GPT-5.2-Codex
Scenario 3	Vibe Coding	Gemini 3.1 Pro

In all three scenarios, the same set of functional requirements was implemented, focused on extending the existing system by associating each film entity with an actor entity – specifically, implementing a many-to-many (M:N) relationship between films and actors with appropriate user interface (UI) support. The requirements included:

- adding a new Actor class,
- explicit implementation of the M:N relationship via a MovieActor join entity,
- seeding initial data such that every film has associated actors,
- displaying a list of all actors,
- displaying each actor’s filmography,
- enabling actor assignment on the film editing component,
- displaying the actor list on the film details component.

In all implementations, the explicit join class for the M:N relationship was deliberately used to avoid differences arising from Entity Framework’s implicit mechanisms.

3.1.1 Manual Development (Scenario 1)

In the first scenario, the entire functionality was implemented manually without working with generative language models. The author possesses foundational knowledge of the Blazor framework and has limited prior experience with this framework. The development workflow involved manual creation of the Actor and MovieActor classes, manual addition of database migrations, seeding logic, creation of new Blazor components for actor display and assignment, and adaptation of existing components.

3.1.2 Vibe Coding (Scenarios 2 and 3)

The remaining two scenarios employed the vibe coding approach using GitHub Copilot in Agent mode. Agent mode enables the autonomous execution of complex development tasks, allowing the agent to analyse existing code, propose and implement changes, and iteratively resolve issues with minimal developer intervention. Unlike standard Copilot usage, Agent mode extends the tool’s role from a coding assistant to a partially autonomous participant in the development process (“About GitHub Copilot Coding Agent”, 2024).

A notable limitation of this study concerns the model selection within GitHub Copilot. As the research was conducted using an educational GitHub Copilot licence, the full range of available models was not accessible; therefore, the models chosen for this study represent the best options available within that constrained environment. GPT-5.2-Codex was selected as OpenAI’s most capable coding-focused model available, purpose-built for long-horizon agentic tasks with support for high-effort reasoning and a 400,000-token context window (“GPT-5.2-Codex Model”, 2025). Gemini 3.1 Pro was selected as a representative model from Google – a different major AI vendor – to enable cross-vendor comparison, as it delivers substantially improved core reasoning over its predecessor, achieving more than double the reasoning performance of Gemini 3 Pro on complex problem-solving benchmarks (“Gemini 3.1 Pro: A smarter model for your most complex tasks”, 2026). In both vibe coding scenarios, identical prompts were provided to GitHub Copilot, instructing it to extend the solution with actor functionality while keeping changes minimal and code clean, adhering to EF Core conventions, and maintaining UI consistency with the existing Blazor application. The role of the author was limited to confirming proposed commands and changes until a successful project build was achieved.

3.2 Evaluation and Code Analysis

Evaluation was conducted using SonarQube Community Edition with the aim of objectively comparing code quality across the three development approaches. The analysis relies exclusively on static code quality

metrics, without additional runtime testing – an acknowledged limitation of the study. The only inclusion criterion was a successful project build, meaning the analysed code had to constitute a deliverable software artefact, which eliminated the need for subjective manual testing from the evaluation criteria.

For each of the three implementations, the following metrics were collected: number of bugs, number of vulnerabilities, number of code smells, cognitive complexity, cyclomatic complexity, number of duplicated blocks and lines, duplicated lines density, number of classes, functions, and files, total lines of code and non-commented lines of source code (NCLOC) and technical debt expressed as the SQALE (Software Quality Assessment based on Lifecycle Expectations) index. Metrics with a value of zero across all implementations (bugs, coverage, line coverage, vulnerabilities) were excluded from the comparative analysis. Due to SonarQube Community Edition's limitations, metric data for each project was retrieved via the SonarQube API and consolidated into a CSV file using a Python script.

4 Results

4.1 Implementation Results by Scenario

In **Scenario 1**, functionality was implemented entirely manually without generative models. Development required a considerable number of iterations, largely due to the author's limited prior experience with the Blazor framework. Multiple errors came up, requiring consultation of documentation and exploration of alternative solutions. The overall development process in this scenario was the longest among all three approaches.

In **Scenario 2**, implementation was carried out using the vibe coding approach with the GPT-5.2-Codex model in GitHub Copilot Agent mode. The agent began by planning out 37 tasks before executing them one by one. Some tasks failed and were retried, and the process involved frequent developer confirmations, making it surprisingly lengthy for a project of this scope. The generated output was nonetheless detailed and required no manual code modifications after the agent finished. A UI issue was observed in the film editing component during manual application launching; however, as runtime testing was outside the scope of the evaluation criteria, this was not further investigated.

In **Scenario 3**, the vibe coding approach was employed with the Gemini 3.1 Pro model. Compared to Scenario 2, the code generation process was faster and required fewer interruptions from the developer, with the agent planning only 5 tasks. The model independently organised the implementation steps, including file creation, modifications to existing code, and addition of accompanying documentation. The project built successfully; however, the application could not be launched immediately, as the code responsible for applying migrations at runtime was missing. The au-

thor used GitHub Copilot's debugger mode to identify the issue, which was then resolved with a fix suggested by the agent.

4.2 Static Code Analysis Results

The results of static code analysis obtained with SonarQube Community Edition are presented in Table 2. Metrics with a value of zero across all three implementations were excluded from the analysis, as described in Section 3.2.

Table 2. Static code analysis results (SonarQube)

Metric	S1	S2	S3
Classes	12	12	11
Code smells	29	15	15
Cognitive complexity	70	64	57
Cyclomatic complexity	219	206	199
Duplicated blocks	2	0	0
Duplicated lines	39	0	0
Dup. lines density (%)	2.2	0	0
Files	34	33	33
Functions	129	122	116
Lines	1751	1719	1681
NCLOC	1140	1107	1084
SQALE index (tech. debt)	172	140	136

All three scenarios produced a comparable amount of code in terms of files, lines, and functions. Scenario 1 recorded the highest values for code smells (29), cognitive complexity (70), cyclomatic complexity (219), and technical debt index (172), as well as the only instances of code duplication, with 2 duplicated blocks across 39 lines at a density of 2.2%. Scenarios 2 and 3 produced identical code smell counts (15) and no duplication, with Scenario 3 showing slightly lower complexity (57 cognitive, 199 cyclomatic) and a marginally lower technical debt index (136) compared to Scenario 2 (64, 206, and 140 respectively).

To complement the static metrics with an indication of development effort, Table 3 reports the approximate development time for each scenario alongside three interaction indicators: the number of tasks the agent planned, the manual code edits required after generation, and the post-build runtime issues encountered. Development time is measured as wall-clock time from the start of the task to the first successful build, including research and debugging. These figures are coarse, single-run observations and are reported as descriptive context rather than as precise productivity measurements.

Table 3. Development effort and interaction indicators per scenario

Indicator	S1	S2	S3
Development time in minutes (approx.)	120-180	30-40	15-20
Agent-planned tasks	—	37	5
Manual code edits after generation	—	0	1
Post-build runtime issues	—	1	1

5 Discussion

The objective of this study was to examine how the application of generative language models and the vibe coding approach affects software code quality in comparison with traditional software development. In addressing RQ1 and RQ2, the literature review demonstrates that existing research largely employs qualitative methodologies and frames vibe coding as a paradigm shift in developer work rather than merely a productivity tool. The empirical results support this perspective: in both vibe coding scenarios, the developer's role changed substantially from that in manual development, with focus shifting from writing code to defining requirements, supervising, and evaluating generated artefacts. Scenarios 2 and 3 are thus consistent with the claim found in the literature that developer work shifts from code authorship toward high-level decision-making, although a single observation per scenario cannot establish this as a general pattern.

Addressing RQ2, the literature warns of the risk of technical debt accumulation caused by accelerated development, insufficiently defined architecture, and vague functional requirements, but prior studies have not used quantitative metrics (van Hurne, 2025; Waseem et al., 2025). The empirical component of this study helps fill that gap through SonarQube analysis. The results show that both vibe coding scenarios produced fewer code smells, lower complexity, and lower technical debt than the manual implementation, with no code duplication present in either. This does not align with the literature's concern about accelerated development leading to increased technical debt – at least not at the level of static metrics on a project of this scope.

Addressing RQ3 directly, for this task and at the level of static metrics, the vibe coding implementations did not exhibit inferior code quality: both Scenarios 2 and 3 scored better than the manual baseline on most measured metrics. This result is specific to the studied task and cannot be generalised to vibe coding as a whole. Scenario 3 (Gemini 3.1 Pro) recorded the lowest values for cognitive complexity, cyclomatic complexity, and technical debt index of all three scenarios, despite requiring a runtime fix after the initial

build. This suggests that the quality of the generated code is not strictly tied to the smoothness of the generation process. However, the runtime issue encountered in Scenario 3 – where migration application code was missing – illustrates that static metrics alone do not capture the full picture of software quality, which is an acknowledged limitation of this study. This aligns with the literature's assertion that generative models do not eliminate the need for programming knowledge, but alter the form in which it is applied – particularly in the phases of evaluation and quality control (Sapkota et al., 2025; Sarkar and Drosos, 2025).

The two vibe coding scenarios also differed in agent behaviour. GPT-5.2-Codex decomposed the task into 37 planned steps and produced a buildable result requiring no manual edits, but left a UI defect; Gemini 3.1 Pro planned only 5 steps, completed faster with fewer interruptions, and achieved marginally better static metrics, yet shipped code that failed at runtime until a migration fix was applied. This contrast suggests a possible trade-off between planning thoroughness and generation speed across vendors, but with a single run per model it is an observation rather than a benchmark, and no claim about the relative capability of the two models is warranted on this evidence.

5.1 Threats to Validity

Before interpreting the results, it is worth being clear about what a study of this design can and cannot establish, since the empirical component is a preliminary, exploratory case study rather than a controlled experiment. The most serious threat concerns internal validity. All three implementations were produced by a single developer with limited prior Blazor experience, so the differences between the manual and vibe coding scenarios cannot be cleanly separated from that developer's learning curve; because the manual scenario was also the first pass through the task, its higher metric values may reflect unfamiliarity as much as the development approach itself. A larger study would need several developers of varying expertise to isolate the effect of the approach from the effect of the person.

External validity is limited in a comparable way. The study rests on a single reference repository (*BlazorWebAppMovies*), a single technology stack (Blazor and Entity Framework Core), and a single, relatively self-contained feature – an M:N extension between films and actors – so the results say little about other languages, frameworks, paradigms, or larger and more architecturally complex systems. The choice of models was further constrained by an educational GitHub Copilot licence, which restricted the comparison to the two available models rather than the most capable models on the market. We worked in this controlled, deliberately simple setting because it lets the three scenarios share identical requirements and a clean baseline; the cost of that control is that the findings do not yet gen-

eralise beyond it.

Construct validity is limited by the decision to operationalise code quality entirely through SonarQube static metrics. These metrics do not capture functional correctness, runtime behaviour, test adequacy, security, or user-facing quality, and the missing-migration defect in Scenario 3 – clean static numbers alongside an application that would not run – makes that gap concrete. With only a single observation per scenario no statistical inference is possible either, so every difference reported here is descriptive rather than confirmatory. Taken together, these considerations are the reason the study is presented as a first, hypothesis-generating step rather than as settled evidence about vibe coding, even as it offers a new empirical perspective on a domain that has so far been investigated largely through qualitative approaches.

6 Conclusion

This paper examined the impact of vibe coding on source code quality through a combination of literature review and static code analysis using SonarQube. The literature review showed that vibe coding is seen in existing research as a shift in how software is developed, with particular emphasis on changes in the developer's role, working practices, and responsibility for code quality. A clear lack of quantitative empirical research was identified as a gap in the current body of knowledge.

The empirical component shows that, for the task studied, the vibe coding implementations did not produce worse code quality in terms of basic static metrics: in both vibe coding scenarios, fewer code smells, lower complexity, and lower technical debt were recorded than in the manual implementation. At the same time, the runtime issue encountered in Scenario 3 shows that good static metrics do not guarantee a fully working solution, and that the developer must remain actively involved in reviewing and validating the generated output.

Based on these results, vibe coding cannot be simply labelled as better or worse than traditional software development. It represents a different way of working that requires new skills and new approaches to quality control. These conclusions are drawn from a single task on a single project and are offered as preliminary, hypothesis-generating evidence rather than as a general finding about vibe coding. The value of this paper lies in providing empirical data where most existing research has relied on qualitative observation, and in connecting theoretical insights with measurable code quality indicators. Among the quality dimensions not captured by static analysis, several stand out as priorities for future work. Functional correctness and runtime behaviour are the most significant: the missing-migration defect in Scenario 3 demonstrates that favourable static metrics can coexist with a non-

functioning application, so verifying that the software actually builds, runs, and behaves as specified is a precondition for any quality claim. Testing quality is a closely related priority, since the present study collected no test-coverage or test-adequacy data, and the degree to which generated code is accompanied by meaningful tests is itself a maintainability concern. Future research should therefore incorporate dynamic evaluation – runtime, integration, and unit testing – together with test-coverage, and extend the analysis to larger and more complex projects and a wider range of generative models.

References

- About GitHub copilot coding agent.* (2024). <https://docs.github.com/en/copilot/concepts/agents/coding-agent/about-coding-agent> (Accessed: 28.04.2026).
- BlazorWebAppMovies sample application.* (2024). <https://github.com/dotnet/blazor-samples/tree/main/10.0/BlazorWebAppMovies> (Accessed: 28.04.2026).
- Brandebusemeyer, C. (2025). Interactions with generative AI: Wearables to measure developer experience and productivity objectively. *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 148–150. <https://doi.org/10.1109/ICSE-Companion66252.2025.00043>
- Casserini, M., Facchini, A., & Ferrario, A. (2026). Beyond the 'diff': Addressing agentic entropy in agentic software development. <https://arxiv.org/abs/2604.16323>
- Chou, Y.-H., Jiang, B., Chen, Y. W., Weng, M., Jackson, V., Zimmermann, T., & Jones, J. A. (2025). Building software by rolling the dice: A qualitative study of vibe coding. <https://arxiv.org/abs/2512.22418>
- Gandhi, D. (2026). Survey on systemic security risks in vibe coding and autonomous development workflows. *TechRxiv*, 2026(0110). <https://doi.org/10.36227/techrxiv.176800890.09196406/v1>
- Gemini 3.1 Pro: A smarter model for your most complex tasks.* (2026). <https://blog.google/innovation-and-ai/models-and-research/gemini-models/gemini-3-1-pro/> (Accessed: 28.04.2026).
- GitHub copilot documentation.* (2024). <https://docs.github.com/en/copilot/get-started/what-is-github-copilot> (Accessed: 28.04.2026).
- GPT-5.2-Codex model.* (2025). <https://developers.openai.com/api/docs/models/gpt-5.2-codex> (Accessed: 28.04.2026).
- Horvat, M. (2026). Which prompting technique is better? Improving vibe coding code quality with efficient prompt design for web front-End development [University of Zagreb, Faculty of Electrical Engineering and Computing]. <https://doi.org/10.5281/zenodo.18165128>

- Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J., & Wang, H. (2024). Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33(8), 1–79. <https://doi.org/10.1145/3695988>
- Jiang, J., Wang, F., Shen, J., Kim, S., & Kim, S. (2025). A survey on large language models for code generation. *ACM Transactions on Software Engineering and Methodology*. <https://doi.org/10.1145/3747588>
- Maes, S. H. (2026, March). *Evaluating the efficacy of artificial intelligence in software engineering: A post-February 2026 analysis* (tech. rep.). <https://doi.org/10.5281/zenodo.19103818>
- Sampson, M. (2025). Enhancing developer experience: Burnout, resilience, and the digital environment in software engineering. *Proceedings of the 38th International BCS Human-Computer Interaction Conference*, 369–374. <https://doi.org/10.14236/ewic/BCSHCI2025.41>
- Sankaranarayanan, S. (2026). Mitigating "epistemic debt" in generative ai-scaffolded novice programming using metacognitive scripts. <https://arxiv.org/abs/2602.20206>
- Sapkota, R., Roumeliotis, K. I., & Karkee, M. (2025). Vibe coding vs. agentic coding: Fundamentals and practical implications of agentic AI. <https://arxiv.org/abs/2505.19443>
- Sarkar, A., & Drosos, I. (2025). Vibe coding: Programming through conversation with artificial intelligence. <https://arxiv.org/abs/2506.23253>
- SonarQube community edition*. (2024). <https://docs.sonarsource.com/sonarqube-cloud/> (Accessed: 28.04.2026).
- van Hurne, M. H. (2025). *A technical debt-aware prompting framework for sustainable vibe coding: Addressing the production readiness crisis in AI-assisted software development* (tech. rep.). TechRxiv. <https://doi.org/10.36227/techrxiv.175459417.76916566/v1>
- Waseem, M., Ahmad, A., Kemell, K.-K., Rasku, J., Lahti, S., Mäkelä, K., & Abrahamsson, P. (2025). Vibe coding in practice: Flow, technical debt, and guidelines for sustainable use. <https://arxiv.org/abs/2512.11922>
- Weisz, J. D., Kumar, S., Muller, M., Browne, K.-E., Goldberg, A., Heintze, E., & Bajpai, S. (2025). Examining the use and impact of an ai code assistant on developer productivity and experience in the enterprise. <https://arxiv.org/abs/2412.06603>