The Gotchas of AI Coding and Vibe Coding. It's All About Support And Maintenance

Stephane H Maes[1]

April 28, 2025

Abstract

This papers reviews AI coding, and in particular the exploding interest in vibe coding, both in terms of main existing framework, advantages and challenges. In particular, we point out in particular an aspect less often discussed: the potential complications for the support and maintenance of software products/code generated via vibe coding. These problems result especially because the generated code often ends up no more be understandable, even to its developers.

Then, we introduce VIBE4M, a framework of workflows, policies and practices to alleviate these challenges. However, such approach goes against the trend that AI makes developers more productive, as they now must perform rigorous code verifications. It also goes against the objective of democratization of coding. Yes coding can be done with "no code", but such code is not maintainable, which may not matter for side projects, but matters for software products. If approaches like VIBE4M are applied, they may be hard to follow for non-programmers. Therefore, there would be value to now automate such frameworks.

____

1. Introduction

AI-based coding was introduced a while ago. In fact, we were among the pioneers putting together such solution. I n our case, it was using natural language instructions to be translated in workflows [14]. At the time, it was using "old AI" NLP techniques for language learning and translation of intent into workflow code. Since, many "studio" AI coding tools, now on LLM-based GenAI, have been introduced, including ChatGPT, GitHub Copilot, Google's Vertex AI Codey and Gemini in Android Studio, IBM's watsonx Code Assistant portfolio, Amazon's Q Developer (formerly CodeWhisperer), Tabnine, Replit Ghostwriter, and Cursor.

The quality of the resulting coding is in the eye of the beholder. Many developers report that they obtained good results when limiting the requests to simple or specific tasks, functions or snippets of code. However, things get rapidly complicated and messy when trying to request to generate in one shot more complex programs [15]. To address these challenges, users may ask the tool to iterate and check and test the code itself, to debug and to correct bugs or to evolve progressively the functionality. Otherwise, in professional settings, developers have to treat the code as if it is produced by a junior programmer, and rigorously review, then test, every piece of code at every step on the way. With Vibe coding, developers can also ask the tool for details on how to compile it and deploy it to run.

[1] shmaes.physics@gmail.com

Cite as: Stephane H. Maes, (2025), "The Gotchas of AI Coding and Vibe Coding. It's All About Support And Maintenance", https://doi.org/10.5281/zenodo.15343349, https://shmaes.wordpress.com/2025/04/28/the-gotchas-of-ai-coding-and-vibe-coding-its-all-about-support-and-maintenance/, April 28, 2025

On one end, these capabilities may appear to be extraordinary. Indeed, with it junior developer can rapidly produce huge amount of code that would otherwise take way longer to generate. Non-programmers can now code, something application that matches what they need as expressed by intents. Productivity seems to shoot up. On the other hand, one need to carefully evaluate the time that it takes to rigorously incorporate AI generated code into a project, then test, debug and deploy it. But for sure, on the surface, it is a great value proposition.

Recently, vibe coding has been promoted as the ultimate way to code without writing a line of code by carrying a conversation with AI tools, directing them to code and correct and misbehavior, till it produces the right result, deploys, and runs seemingly seamless [14,16]. It is not just that one can easily get help in coding, but rather that one really can just delegate and guide coding to the tools. Fantastic isn't it? Especially with testimonials that it seems that LLMs, agents and AI tools have reached a level where it is possible to make it all work [14].

Non-developers seem to now be able to code rapidly and efficiently, even without actually writing a line of code. They are now able to generate the code required to support a particular use case or workflow, on their platform of choice, or in their programming language of choice. There is no doubt that it is fantastic for non-developers, who can now build an "app" without even knowing what they are doing, just knowing what they want to see the app do. Developing has now been democratized!

Similarly, executives in many companies, including technology savvy companies or software companies, see AI-coding as an opportunity to dramatically increase productivity, and ultimately, after complimenting, to actually eliminate many developer jobs [19], especially senior roles who command higher salary. It may be wishful thinking, but that may be discovered too late as we will discuss. Developers will still be needed, unless and until frameworks like VIVE4M, discussed in this paper, are automated or autonomous.

Furthermore, as we stated, the quality of AI-generated code is still questionable. Even, the original tweet promoting vibe coding [14] admitted that, at the end of the process, everything can go bad, or result in working code that even an experienced developer cannot understand any more. Again, it may not matter if you are a non-developer, and want just to develop a custom app right now for a specific problem.

But it is more problematic for software products and enterprises. Indeed, it may not work if your ambition is to create a supported product with a roadmap of releases and features. Sure, one may hope that giving the previously released code another pass of vibe coding can:

- 1) fix any problem that support and maintenance has identified
- 2) suitably add new requires capabilities or support of other non-functional requirement, all while ensuring suitable backward compatibility, confirmed by appropriate regression tests[2].

Think of maintaining outsourced code, or escrowed code, etc., which already would have the advantage over vibe code that it would be understandable to humans. It is not an easy task and often requires full reverse engineering. Can vibe coding, or other AI coding tools, do that? At least as a reasonable process. Alternatively, is there a way to use vibe coding, or extend its "processes" to address production ready maintenance and support?

In the paper., we discuss these options, noting that some of them imply keeping an experienced development team along, i.e., no more thinking of getting rid of senior developer positions. Also, some of these processes are just not something that we can expect to be easily performed by non-technical developers, which ay or may not rain on the parade of democratizing coding.

After reviewing AI coding tools and the challenges with vibe coding, we will discuss best practices, and propose VIBE4M as a framework for vibe coding while addressing, or at the minimum mitigating these challenges, focused

[2] While the former (fixing by vibe coding) may be achievable, good luck to make it understand, teste and ensure backward compatibility.

on making the code supportable and maintainable. We invite feedback (comment on web site [18], or email to author) from those who select to implement VIBE4M. We plan to publish further papers on automating VIBE4M, if possible towards an autonomous framework.

In the meanwhile, we hope that reader will heed our advice to take seriously the risks of simply using Vibe coding without careful ways to mitigate these risks. Not doing so can lead to catastrophes [25].

2. AI development Tools

The landscape of software development is undergoing a significant transformation, driven largely by the advent and rapid maturation of Artificial Intelligence (AI). AI-powered coding assistants have emerged as pivotal tools, moving beyond simple autocompletion to offer sophisticated capabilities that augment developer workflows, enhance productivity, and influence the entire software development lifecycle (SDLC). These tools leverage large language models (LLMs) and machine learning techniques to understand code context, generate suggestions, translate natural language into code, identify bugs, assist with documentation, and even automate complex refactoring tasks. As organizations seek to accelerate development cycles, improve code quality, and empower their engineering teams, understanding the capabilities, strengths, and strategic positioning of the leading AI coding assistants becomes crucial.

In this paper, our analysis relies on experimenting and studying prominent AI coding tools, including GitHub Copilot, Google's Vertex AI Codey and Gemini in Android Studio, IBM's watsonx Code Assistant portfolio, Amazon's Q Developer (formerly CodeWhisperer), Tabnine, Replit Ghostwriter, and Cursor, examining their features, target audiences, and the evolving dynamics of the AI-assisted coding market.

At the time of publication of this paper, the market for AI coding assistants features a diverse range of tools, each with distinct approaches and target markets:

- GitHub Copilot: Developed via a collaboration between GitHub, OpenAI, and Microsoft, Copilot is one of the most widely adopted tools, known for its deep integration into the GitHub ecosystem and strong code completion capabilities [3].
- Google (Vertex AI Codey & Gemini in Android Studio): Google offers specialized AI tools. Vertex AI Codey provides APIs for code generation, completion, and chat, often integrated via Google Cloud [4]. Gemini focuses specifically on enhancing Android development within Android Studio [5].
- IBM (watsonx Code Assistant & watsonx Code Assistant for Z): IBM targets enterprise needs, offering a general watsonx Code Assistant for various languages and a specialized version, watsonx Code Assistant for Z, focused on mainframe (COBOL, PL/I, JCL) application modernization [6-8].
- Amazon Q Developer (formerly CodeWhisperer): Amazon's offering, now part of the broader Amazon Q suite, emphasizes integration with AWS services, security scanning, and assistance beyond coding into cloud operations [9].
- Tabnine: This assistant prioritizes privacy, security, compliance, and personalization, offering flexible deployment options (including self-hosting) and the ability to train custom models on private codebases [10].
- Replit Ghostwriter: Tightly integrated into the Replit online IDE, Ghostwriter targets education, learning, and collaborative cloud-based development within the Replit ecosystem [11].

- Cursor: An AI-first code editor built on VS Code, Cursor aims for deep AI integration into the core editing experience, offering features like predictive multi-line edits and codebase-aware chat [12].
- ChatGPT and other LLMs, conversationally requested to code.

These tools represent different philosophies, ranging from general-purpose IDE plugins, and platform-specific enhancements, to specialized enterprise solutions and entirely new editor paradigms.

3. Vibe Coding

Vibe coding was popularized by a tweet [14]. It is an AI-assisted approach where the developer describe his or her software idea in plain natural language, and the AI writes the code. We implemented an early version with old AI, i.e., not LLM-based GenAI, 9 years ago, and with one shot natural description of the desired intent [13].

So, vibe coding refers to a prompt-driven approach to software development where developers describe what they want in natural language, letting an AI generate the code rather than writing it manually [16]. This development paradigm represents a significant shift in how software is created, emphasizing intent over implementation details, as already proposed in [13][3].

In vibe coding, developers provide short instructions, or "vibes" (often just a sentence or two), rather than detailed specifications[4], and the AI translates these ideas into functional code. This process shifts the developer's focus from "how" software should be implemented to "what" it should accomplish, potentially democratizing coding and speeding up development.

The developer becomes more of a director, guiding the AI through prompts, rather than manually crafting each line of code. As one analysis notes:

Skillful prompting is required to get good results. Ambiguous or overly broad instructions often lead to messy or incorrect code. You have to learn to 'talk' to the AI effectively, much like guiding a junior programmer.

It is not hard to see that this opens the door “no code” coding, i.e., coding without writing a single line of code, and this democratized on step further the ability for anyone to develop “apps”. It is a fantastic and revolutionary development for simple or one-off projects. But is it a sustainable and manageable approach for complex projects and products?

3.2 Vibe Coding Options and Approaches

Several methodologies have emerged for effective vibe coding implementation:

[3] [13] proposed it as an initial full intent description, rather than a conversation of short prompts. Assembly was left to the developer or tool.
[4] [13] was rather the latter, but the principle was to do that small task by small task (as it was done for workflows), so that one prompt was enough.

- Vibe PMing: Starting projects by having AI create a readme that includes requirements, tech stack, and development milestones [17]. This "product management" phase establishes boundaries before actual coding begins.
- Simplified Tech Stacks: Keeping technology choices simple to reduce the likelihood of AI breaking applications. Many developers use client-side technologies with local storage rather than complex server architectures [17].
- Stepwise Development: Breaking complex tasks into small, manageable steps to maintain control over the development process and prevent AI from generating overly complicated solutions [17].
- Asking AI to explain and comment is essential, but unfortunately often ignored, or an afterthought. We will come back to this.

Early experiences suggest vibe coding is particularly effective for:

- Rapid prototyping and MVPs (Minimum Viable Products) [18]
- Projects with limited lifespans, or maintenance requirements
- Educational contexts, and non-critical applications
- Situations where accessibility to non-developers is prioritized [19]

As an example, Replit states that 75% of Replit users never write a single line of code. They simply describe what they want, and AI builds it for them [19].

4. Challenges with Vibe Coding

4.1 Code Quality and Technical Debt

We can raise legitimate concerns about code quality with vibe coding approaches. AI-generated code, while functional, often lacks the thoughtful architecture, and dedication to technical excellence, that experienced developers bring to projects [18]. The maintenance issues manifest in several specific ways:

- Implementation Fragility: Vibe-coded solutions often work for specific use cases but break when conditions change, creating brittle systems [18].
- Architectural Inconsistency: AI generates solutions based on different prompts, with the risk of lacking a unified vision, creating a patchwork codebase where similar problems are solved in dissimilar ways [20]. In fact, besides duplication of functionality and inconsistent architecture, one can expect that this risk to result into a very poor user experience where the user becomes aware that certain behaviors change depending on where they are encountered. It has been argued that vibe coding leaves time for developers to become software architects. Except that it requires different expertise, and it rarely works with junior developers. It certainly does not help non-programmers.
- Technical Debt Acceleration: The process of vibe coding has been compared to "taking out a high-interest loan against your codebase's future." The immediate productivity gains feel valuable, but eventually, the

technical debt payments come due in the form of maintenance challenges, refactoring needs, and scaling or enhancement difficulties.

As a result, support and maintenance can become a nightmare. AI-generated code often lacks the structure, documentation, understanding of the code, and clarity necessary for long-term maintenance. It can lead to increased technical debt, making support, enhancement, future modifications, regression testing and debugging significantly more difficult, potentially requiring costly rewrites.

4.1.1 Mathematical Model of Technical Debt Accumulation

We can model the accumulation of technical debt in vibe coding projects using a simplified equation:

$$TD(t) = \sum_{i=1}^{n} (C_i \times (1 - R_i) \times e^{kt}) \quad (1)$$

Where:

- $TD(t)$ represents technical debt at time t
- C_i represents the complexity of feature i
- R_i represents the review/refactoring effort applied to feature i (between 0 and 1)
- k is a constant representing the exponential growth rate of technical debt
- n is the number of features

This model illustrates how technical debt compounds exponentially without adequate review and refactoring, particularly in vibe coding environments where R_i tends to be lower than in traditional development.

4.2 Debugging and Troubleshooting Difficulties

Debugging AI-generated code presents unique challenges that compound over time. Vibe coding is all fun and games until you have to vibe debug [18].

The debugging difficulties include:

- Lack of Developer Comprehension: When developers don't fully understand how the generated code works, identifying root causes becomes exponentially more difficult [21]. Even more when this is to be done later in past versions of a program.
    - The fact that we repeatedly see reports that developers using vibe coding may lose understanding of the resulting code after a few iterations of code refinement, debugging or inclusion in a wider program, is really the most problematic part. Having experienced the challenge in up taking, or reverse engineering, escrowed code, which was traditionally developed, and hence more easily understood by humans, we can only imagine the challenges of repeating the task with code not even (well) understood by its original developer(s).

- Trial-and-Error Debugging Loops: Inexperienced developers often fall into a pattern of prompting the AI repeatedly with slight variations, accepting unverified fixes, and failing to trace root causes manually [21].
- Reactive vs. Systematic Troubleshooting: Unlike traditional debugging, where developers analyze call stacks, state transitions, and system logs, vibe coding often reduces the process to guesswork, or, depending on the AI, to fix its own mistakes [21].
    - Imagine now doing that on earlier on versions of such code. It always amounts to having lost the original development team, and having nobody understanding the code, and the details of the architecture of the product. Unless of course if rigorous processes were followed, as sketched in [17], for example, or later in the paper. Butt are these enough, and will they be easy to follow?

It is no surprise that Microsoft engineers recently noted that AI-generated projects struggle with long-term maintainability, making vibe coding a poor fit for enterprise software [19].

4.3 Security Vulnerabilities

Security represents another major concern in vibe-coded projects. AI coding assistants primarily optimize for functionality rather than security, potentially introducing vulnerabilities that (experienced) human developers would hopefully quickly identify [18]. AI-assisted code can increase security risks in complex systems [21].

These security issues stem from several factors:

- Optimization Bias: AI tools prioritize making code work over making it secure
- Blind Spots: AI doesn't "think" like attackers, and may miss common vulnerability patterns. It also may not know how the code it develops fits or does not fit vulnerability patterns and best practices.
- Dependency Issues: Generated code often includes unnecessary, or outdated, dependencies with known security flaws. After all, coding by LLM was learned on past code, which may carry forward all its flaws, bugs, incorrect, or outdated, patterns and vulnerabilities.

Industry response to these concerns is emerging, with security companies like Snyk offering workshops specifically on uncovering vulnerabilities in AI-assisted coding [22], or with AI models trained to address secure code generation [b23], which perform better but still fails too often.

4.4 Documentation and Knowledge Transfer Problems

Documentation suffers significantly in vibe coding environments. The focus shifts to prompt engineering rather than explaining code functionality, creating code that no one can explain. It is a recurrent theme, never encountered in the past, where there was always somebody, maybe confused and hard to follow, or having challenges to explain, but who knew what was going on.

Conventional coding already does not include good documentation, and requires best practices and processes to ensure that comments and documentation move along. Even with all this, documentation of conventional coding has always been a weakness. So, imagine what it means with code that is not understood by its coders…

Even if comments, and documentation. were systematically generated, with the help of AI, what is to say that these will be correctly characterizing the code instead of possibly confused, incorrect, misleading or useless. After all, next to hallucinated references, hallucinated comments are to be expected [6,94].

Any documentation gap creates significant challenges for:

- Onboarding: New team members struggle to understand systems without clear documentation
- Knowledge Transfer: When the original prompter leaves, vital context is lost
- Maintenance: Future modifications become high-risk without understanding original intent
- Support of existing and past releases
- Enhancement of the code

Again, these challenges can only be further exacerbated with Vibe coding.

4.5 QA testing

Quality Assurance (QA) for vibe-coded applications presents unique challenges that extend beyond traditional software testing methodologies. The spontaneous nature of AI-assisted code generation introduces significant verification hurdles, as recent studies indicate leading AI coding tools like ChatGPT and GitHub Copilot produce correct code only 65.2% and 46.3% of the time respectively [25,26].

This imperfect generation creates code that looks like a house of cards code, i.e., implementations that function superficially but harbor critical deficiencies in error handling, performance optimization, and security compliance [24]. The testing complexity is further magnified by accountability issues, as developers tend to scrutinize AI-generated code less rigorously than their own creations, feeling diminished responsibility for its quality [26]. This reduced ownership contributes to consistent outages in production, directly attributed to inadequately tested AI-generated implementations [25].

The reproducibility challenge presents another obstacle, as stochastic aspects elements in all aspects of AI generation create inconsistent outputs even with identical inputs, complicating test case design and validation processes [27].

Additionally, the rapid iteration cycles, characteristic of vibe coding often bypass comprehensive testing frameworks, result in applications with structural inconsistencies and poor maintainability, which resist conventional automated testing approaches [28]. These challenges necessitate specialized testing environments that can effectively simulate the diverse operational conditions under which vibe-coded applications must function reliably [29].

The lack of understanding of the code makes it hard to document, and test the software, and then debug it. It complicates the development of test cases, or automation. It also makes test engineer uncomfortable, as they do not under stable aspects of what they test.

Of course, one may decide to also vibe test the vide-coded code. Now the problem is how to trust and understand the testing environment and automations, that would, in alignment with vibe coding, be AI generated and changing

all the time[5]. Are we sure they test what is to be tested? Again the lack of understanding of the code means lack of understanding of what exactly is being tested.

5. Comparative Analysis: Vibe Coding vs. Structured Development

The following table outlines key differences between vibe coding and structured development practices across various dimensions:

Aspect	Vibe Coding	Structured Development
Approach	Ad-hoc & prompt-driven: Code is generated on-the-fly by an AI from natural language prompts, with minimal upfront design	Planned & design-driven: Code is written according to pre-thought design or methodology (e.g., TDD, DDD)
Speed of Development	Very fast initial output: AI can produce large chunks of code in seconds, enabling rapid prototyping	Deliberate pace: Slower to start since humans must write code and tests manually
Code Quality	Variable/Unpredictable: Quality depends on AI training and prompt guidance. May be functional but not optimal or maintainable	Emphasized from start: Developers apply best practices, design patterns, and refactoring to ensure code clarity and proper architecture
Debugging	AI-assisted trial & error: When errors occur, typical response is to feed the error back to the AI or tweak prompts until it works	Systematic troubleshooting: Developers use debuggers, log analysis, and reasoning to locate root causes
Testing	Often omitted or added late: Tests come after code and often after bugs appear	Integral to process: Tests are a standard step, often preceding implementation in TDD approaches
Maintainability	Potentially low: Without structure and human understanding, code can be brittle and difficult to modify	High (by design): Code is crafted to be readable and modular, following established principles like SOLID

Table 1. Comparison (based on [21]) between Vibe coding and best practices-based conventional structure coding

[5] Think of the challenges in regression testing when the code is later modified. Will the vibe testing suite still be adequate. If they are modified do they really provide regression testing?

The comparison highlights the fundamental trade-off: vibe coding offers remarkable speed and accessibility, but sacrifices the architectural integrity and maintainability that structured development prioritizes.

6. Best Practices Identified So Far For Sustainable Vibe Coding

This section summarizes some of the early best practices identified in developers' communities, to address the challenges above. These apply not just to vibe coding, but to AI coding in general (and of course any coding practices including conventional ones as well).

6.1 Code Review and Quality Control

To mitigate the maintenance challenges of vibe coding, several best practices have been suggested:

- Mandatory Code Review: "Always Review AI-Generated Code" with no exceptions. Every block of code that AI produces should be treated as if a junior engineer wrote it. If you don't have time to review it, you don't have time to use it [24].
- Established Coding Standards: Define team style guides, architecture patterns, and best practices, ensuring that AI-generated code is refactored to comply. For instance, if your rule is "all functions need JSDoc comments and unit tests," then AI output must get those comments and tests before it's considered complete [24]. And code review should review them.
- Automated Quality Checks: Create linting, or static analysis checks specifically for common AI mistakes, such as flagging use of deprecated APIs, or overly complex functions. This automates quality control on AI contributions [24]. Automated pen testing and review of secure patterns / know vulnerabilities should be considered.
- The Explainability Rule: "Don't commit code you couldn't explain to a teammate." This reintroduces structure into the workflow, transforming vibe coding into standard development with AI as a tool, not a crutch [21].

6.2 Architectural Guidelines

To address architectural concerns in vibe-coded projects, organizations should:

- Maintain Architectural Discipline: Ensure AI-generated code follows the project's established architecture rather than allowing drift
- Refactor Regularly: Schedule refactoring sessions to resolve architectural inconsistencies before they compound

- Feature-Architecture Balance: Balance feature-driven development (common in vibe coding) with architectural considerations

Without these guardrails, vibe-coded applications often resemble a patchwork of disconnected solutions, while structured systems are designed for cohesion and extensibility from the beginning [21].

6.3 Testing Strategies

Effective testing becomes even more crucial for maintaining vibe-coded projects:

- Test-Driven Development: Consider writing tests before generating code to ensure AI output meets specific requirements
- Comprehensive Coverage: Implement unit, integration, and system tests to catch subtle bugs introduced by AI
- Regression Testing: Maintain robust regression tests to prevent AI-introduced changes from breaking existing functionality

6.4 Overall

Vibe coding represents a significant paradigm shift that offers remarkable development speed and accessibility, but introduces substantial maintenance and support challenges. The evidence suggests that these approaches are most suitable for prototyping, MVPs, and non-critical applications rather than enterprise systems or projects requiring long-term maintenance.

For organizations adopting vibe coding, establishing disciplined processes around code review, architectural guidelines, testing, and documentation is essential for mitigating long-term maintenance costs. The most sustainable approach appears to be a hybrid methodology that leverages AI for initial development speed while incorporating structured practices to ensure maintainability.

Vibe coding enables rapid prototyping, but it lacks the safeguards of structured development. When used without discipline, it can lead to fragile, opaque, and unscalable systems. For serious projects, integrating AI into intentional and reviewable workflows is the only sustainable approach [21].

7. VIBE4M: A Framework for Ensuring Maintainability in AI-Assisted Code Generation

Let us now deep dive into the support and maintainability challenge and possible solution.

So far, as we have seen, vibe coding is a recently emerged paradigm where developers use AI to generate code through natural language prompts, that is considered to offer significant advantages in development speed and accessibility. However, this approach introduces substantial maintenance and support challenges that must be addressed.

This section introduces VIBE4M (Verifiable Inspection-Based Enhancement for Maintainability), a comprehensive framework addressing the specific maintainability challenges of AI-generated code while preserving productivity benefits.

7.1 Introduction to the Maintainability Challenge in AI-Generated Code

The software development landscape has undergone a paradigm shift with the introduction of AI coding assistants. While these tools significantly accelerate development cycles and democratize programming, they raise substantial concerns about the maintainability of the resulting code. Code written with AI is often harder to maintain and of lower quality as it's often verbose or copy-pasted" [30]. By comparing human-written and AI-generated solutions, it is clear that the generated code is harder to alternate/modify/enhance to support new properties, while the human created code is much easier.

The maintainability challenge becomes particularly acute when developers integrate AI-generated code without fully understanding its implementation. We already quoted that vibe coding is all fun and games until you have to vibe debug [18]. This creates a scenario where, initially, the code functions correctly, but it becomes increasingly difficult to modify, extend, or troubleshoot over time.

7.1.1 Understanding Code Maintainability

Maintainability is fundamentally a measure of how easily code can be understood, corrected, adapted, and enhanced. It involves factors such as code readability, modularity, and documentation [d3]. In organizational contexts, maintainability requires organization-wide coordination, since it relies on being able to search, reuse, and change other teams' code"[32].

The significance of code maintainability extends beyond immediate development concerns. High maintainability correlates with:

- Reduced long-term development costs and reduction of technical depth
- Improved ability to implement new features, and enhancement requests
- Lower defect rates
- Ability to support past versions
- Ease to implement regression testing
- Enhanced knowledge transfer between team members
- Greater resilience to developer turnover

7.1.2 Traditional Approaches to Maintainability Assessment

Software maintainability has traditionally been assessed through various methods [33]:

- Hierarchical multidimensional assessment models that "view software maintainability as a hierarchical structure of the source code's attributes"
- Polynomial regression models used to "explore the relationship between software maintainability and software metrics"
- Aggregate complexity measures that gauge "software maintainability as a function of entropy"
- Principal components analysis for statistical evaluation

These approaches have proven effective for conventional development methodologies, but require adaptation for AI-assisted coding environments.

7.2 The VIBE4M Framework

Our proposed VIBE4M framework addresses the unique maintainability challenges of vibe coding through a comprehensive, multi-layered approach integrating both automated and manual techniques. The framework consists of several interconnected components, described in the following subsections.

7.2.1 AI-Aware Code Inspection Protocol (ACIP)

The ACIP extends traditional code review practices with specific considerations for AI-generated code. It introduces:

- AI Origin Tagging
    - Automatic identification and tagging of code sections generated by AI tools, and meta data about which tool.
- Comprehension Verification: Explicit verification that human developers understand the AI-generated code. This includes comments in the code and explainability of the code and may refer to documentation generated on the side.
- Refactoring Recommendations: Targeted suggestions for improving maintainability of AI-generated sections.

The protocol implements a dual-review system where reviewers must explicitly confirm their understanding of AI-generated code segments before approval.

We then have:

$$M_{understanding} = \frac{\sum_{i=1}^{n} C_i}{n} \geq T_{threshold} \quad (2)$$

Where:

- $M_{understanding}$ represents the measured understanding level
- C_i represents the comprehension score for each code segment
- n is the number of AI-generated segments
- $T_{threshold}$ is the minimum acceptable understanding threshold

7.2.2. Explainability Enhancement Layer (EEL)

The EEL addresses the opacity of AI-generated code by requiring and generating explicit documentation of design decisions and implementation rationale:

- Context Preservation: Capture and preservation of the prompts and constraints that generated the code
- Decision Tree Documentation: Visual representation of the AI's decision-making process
- Alternative Implementation Comparison: Documentation of alternative approaches considered by the AI, produced by the Ai system.

This greatly benefits from using a vibe coding system, i.e., set of tools, which relies on explainable AI [35-d37]. So far this has not been a focus of these tools, and there are, to our knowledge, no actual AI coding tool, or vibe coding tool providing explainability. It is important that this be not limited to local functions but also to the overall workflows calling these functions.

7.2.3. Auto-Generated Documentation Verification (AGDV)

This component ensures that AI-generated code includes accurate and comprehensive documentation:

- Documentation Coverage Analysis: Automatic assessment of documentation completeness
- Natural Language Validation: Verification that documentation is understandable by humans
- Consistency Checking: Automated verification that documentation accurately reflects code behavior

Maintainability is the metric for how easy it is to keep documentation accurate, relevant, and up-to-date [34]. The AGDV component leverages this insight to create a continuous validation loop:

```
function AGDV_Score(codeModule) {
  coverageScore = assessDocumentationCoverage(codeModule)
  readabilityScore = assessDocumentationReadability(codeModule)
  consistencyScore = verifyCodeDocConsistency(codeModule)

  return (0.4 * coverageScore + 0.3 * readabilityScore + 0.3 * consistencyScore)
```

```
}
```

Ideally measures should be taken to ensure that the documentation matches the code, as part of the QA process.

7.2.4. Controlled Integration Workflow (CIW)

The CIW establishes a structured process for incorporating AI-generated code into production systems through:

- Graduated Integration: Progressive incorporation of AI-generated code in increasing levels of system criticality
- Maintainability Gates: Specific thresholds for maintainability metrics that must be met before integration
- Contextual Tagging: Preservation of metadata about generation context for future reference

This workflow addresses the observation that "vibe coding is fantastic for MVP, but more frustrating for rewrites in larger code bases" [18] by ensuring that only maintainable code advances to production.

7.2.5. AI-Specific Maintainability Metrics (ASMM)

The ASMM component introduces novel metrics specifically designed to assess the maintainability of AI-generated code:

- Comprehension Effort Index (CEI): Measures the cognitive effort required to understand AI-generated code
- Modification Complexity Ratio (MCR): Compares the complexity of modifying AI-generated vs. human-written code
- Documentation-Implementation Alignment (DIA): Assesses the alignment between documentation and implementation

The CEI is calculated using:

$$CEI = \alpha \cdot C_{cyclomatic} + \beta \cdot D_{naming} + \gamma \cdot S_{structural} \quad (3)$$

Where:

- $C_{cyclomatic}$ represents cyclomatic complexity
- D_{naming} measures naming convention deviation
- $S_{structural}$ quantifies structural complexity
- α, β, and γ are weighting factors

7.3 Implementation Strategy of VIBE4M

Implementing VIBE4M requires integration into existing development workflows and toolchains. The framework can be introduced gradually through the following phases:

Phase 1: Assessment and Baseline Establishment

- Conduct maintainability audits of existing codebase using automated tools
- "Static Code Analysis identifies code with high complexity (cyclomatic complexity or deep inheritance)"
- Establish baseline maintainability metrics for comparison

Phase 2: Tool Integration and Process Adaptation

- Integrate ACIP into existing code review processes
- Implement EEL through AI prompt engineering and documentation tools
- Deploy AGDV as part of the continuous integration pipeline

Phase 3: Metric Collection and Workflow Enforcement

- Begin collecting ASMM data to establish organizational benchmarks
- Enforce CIW processes for all new AI-generated code
- Monitor and adjust thresholds based on team feedback and performance

Phase 4: Continuous Improvement and Adaptation

- Refine metrics and processes based on collected data
- Adjust thresholds and weightings to suit specific organizational contexts
- Conduct regular training to enhance developer awareness of maintainability concepts

7.4 Validation Methodology

To validate the effectiveness of the VIBE4M framework, we propose a three-part validation methodology:

1. Controlled Comparative Studies

Compare maintainability outcomes between codebases developed with and without the VIBE4M framework through:

- Independent expert evaluation of code quality
- Measurement of time required for feature addition

- Analysis of defect rates in maintained code

2. Metrics Analysis

Track key maintainability metrics over time to assess the long-term impact of VIBE4M:

- Technical debt accumulation rate
- Time required for onboarding new developers
- Frequency and complexity of maintenance issues

3. Developer Experience Assessment

Evaluate the subjective experience of developers working with VIBE-M through:

- Structured interviews and surveys
- Cognitive load measurements during maintenance tasks
- Time tracking for maintenance activities

We welcome feedback from teams or organizations deploying VIBE4M.

7.5 Projected Outcomes and Benefits of VIBE4M

Implementation of the VIBE4M framework is expected to deliver several key benefits:

- Enhanced Code Understanding: Developers gain deeper comprehension of AI-generated code through explicit explanation requirements
- Reduced Technical Debt: Systematic identification and remediation of maintainability issues in AI-generated code
- Improved Knowledge Transfer: Better documentation and explainability facilitate knowledge sharing between team members
- Accelerated Maintenance Activities: Clear structure and comprehensive documentation reduce time required for maintenance
- Balanced Innovation and Stability: Organizations can leverage AI productivity benefits while maintaining sustainable codebases

7.6 Status

The VIBE4M framework represents a holistic approach to addressing the unique maintainability challenges posed by AI-assisted code generation in terms of support and maintenance. By combining established software

engineering practices with novel techniques specifically designed for AI-generated code, VIBE4M enables organizations to harness the productivity benefits of vibe coding while ensuring long-term code maintainability.

Organizations willing to thoughtfully integrate vibe coding into their development workflows, recognizing where it adds value and where traditional approaches remain superior, stand to gain significant advantages in speed and accessibility" [18]. The VIBE4M framework provides precisely this thoughtful integration approach, enabling sustainable development practices in the age of AI-assisted coding.

However VIBE4M is a framework with a set of processes, rules and guidelines, or policies. They are not automated yet, and for sure require more time and implementation examples to be confirmed as valuable. That is why we invite feedback from organizations or teams willing to try it.

In an upcoming paper, we will describe an automated, and somehow autonomous, implementation with rules and AI of such a vibe coding framework as VIBE4M.

8. Conclusions

In this paper, we reviewed vibe coding and then we established the main challenges with AI coding, and in particular vibe coding. We also discussed possible recommendations or best practices to alleviates the challenges with vibe-coded software in terms of support and maintenance.

We proposed VIBE4M as a framework of rule, practices and tools to follow these recommendations. In future papers, we will discuss ways to automate such frameworks.

And yes, we were pleased to show that [13] anticipated much of vibe coding. More constrained, it did not have the same support and maintainability challenges.

———

References

[1]: Stephane H. Maes, (2024), “The Trouble with GenAI: LLMs are still not any close to AGI. They will never be”, https://zenodo.org/doi/10.5281/zenodo.14567206, https://shmaes.wordpress.com/2024/12/26/the-trouble-with-genai-llms-are-still-not-any-close-to-agi-they-will-never-be/, December 25, 2024. (osf.io/qdaxm/, viXra:2501.0015v1).

[2]: Stephane H. Maes, (2024), “Fixing Reference Hallucinations of LLMs”, https://doi.org/10.5281/zenodo.14543939, https://shmaes.wordpress.com/2024/11/29/fixing-reference-hallucinations-of-llms/, November 29, 2024. (osf.io/u38w4/, viXra:2412.0149v1).

[3]: “GitHub Copilot · Your AI pair programmer”, GitHub, https://github.com/features/copilot. Retrieved on April 24, 2025.

[4]: “Use AI to generate code with human language prompts”, Google, https://cloud.google.com/use-cases/ai-code-generation. Retrieved on April 26, 2025.

[5]: “Gemini in Android Studio”, Google for Developers, https://developers.google.com/gemini-code-assist/docs/android-studio-overview. Retrieved on April 24, 2025.

[6]: “Watsonx”, IBM, https://www.ibm.com/watsonx. Retrieved on April 24, 2025.

[7]: “IBM watsonx Code Assistant for Z”, IBM, https://www.ibm.com/products/watsonx-code-assistant-z. Retrieved on April 24, 2025.

[8]: “IBM watsonx Code Assistant”, IBM, https://www.ibm.com/products/watsonx-code-assistant. Retrieved on April 24, 2025.

[9]: “What is CodeWhisperer?” - AWS Documentation, AWS, https://docs.aws.amazon.com/codewhisperer/latest/userguide/what-is-cwspr.html. Retrieved on April 24, 2025.

[10]: “Any agent can write code. Ours earn your devs' trust.”, Tabnine AI Code Assistant, https://www.tabnine.com/. Retrieved on April 24, 2025.

[11]: “Intro to Ghostwriter”, Replit, https://replit.com/learn/intro-to-ghostwriter. Retrieved on April 24, 2025.

[12]: “The AI Code Editor”, Cursor, https://www.cursor.com/. Retrieved on April 24, 2025.

[13]: Stephane H Maes, Karan Singh Chhina, Guillaume Dubuc, (2016-2021), “Natural language translation-based orchestration workflow generation”, US 11,120,217 B2.

[14]: Andrej Karparthy, (2025), https://x.com/karpathy/status/1886192184808149383, February 2, 2025.

[15]: /r/CursorAI, (2025), “My honest review after 3 months with CursorAI: Don’t use it. My honest review after 3 months with CursorAI: Don’t use it — unless you really know what you’re doing.”, Reddit, https://www.reddit.com/r/CursorAI/comments/1k3uz9k/my_honest_review_after_3_months_with_cursorai/. April 2025.

[16]: Nate B. Jones, (2025), “The Vibe Coding Bible: How to Build Useful Things with Short Prompts. Vibe Coding is all the rage since Andrej coined the term a month ago, but what does it really mean? How can you build useful things with it? What are the principles and pitfalls? Is it just a gimmick?”, https://natesnewsletter.substack.com/p/the-vibe-coding-bible-how-to-build, March 12, 2025.

[17] Peter Yang, (2025), “12 Rules to Vibe Code Without Frustration. My best tips after 50+ hours building apps with AI (without knowing how to code)”, https://creatoreconomy.so/p/12-rules-to-vibe-code-without-frustration, March 19, 2025.

[18]: Stefan Wolpers, (2025), “Is Vibe Coding Agile or Merely a Hype?”, Age of Product, https://age-of-product.com/vibe-coding-agile/, March 23, 2025.

[19]: Vibe code carrers, (2025), “Vibe Coding vs. Traditional Coding: Pros & Cons”, https://www.vibecodecareers.com/blog/vibe-coding-vs-traditional-coding, March 13, 2025.

[20]: Federico Trotta, (2025), “5 Vibe Coding Risks and Ways to Avoid Them in 2025. Explore the dangerous vibe coding risks that come with technology revolutions and learn actionable prevention strategies to protect your codebase.”, Zencoder, https://zencoder.ai/blog/vibe-coding-risks, April 2, 2025.

[21]: Alesia Sirotka, (2025), “What’s The Problem With Vibe Coding?”, Flat logic, https://flatlogic.com/blog/what-s-the-problem-with-vibe-coding-honest-review/, April 14, 2025.

[22]: Jonathan Santilli, (2025), “The Hidden Dangers of Vibe Coding”, Dev, https://dev.to/pachilo/the-hidden-dangers-of-vibe-coding-3ifi, March 19, 2025.

[23]: Emma Woollacott, (2025), “Want to supercharge your vibe coding skills? Here are the best AI models developers can use to generate secure code. Claude 3.7 Sonnet is the best performer for vibe coding, while others

produce very mixed results", IT.Pro, https://www.itpro.com/software/development/vibe-coding-best-ai-models-secure-code-generation, April 25, 2025.

[24]: Addy Osmani, 920250, '' Vibe Coding is not an excuse for low-quality work. A field guide to responsible AI-assisted development '', Substack, https://addyo.substack.com/p/vibe-coding-is-not-an-excuse-for, April 17, 2025.

[25]: Fiona Jackson, (2024), "AI-Generated Code is Causing Outages and Security Issues in Businesses", Tech Republic, https://www.techrepublic.com/article/ai-generated-code-outages/, September 13, 2024.

[26]: Sean Michael Kerner, (2025), "The risks of AI-generated code are real - here's how enterprises can manage the risk", VentureBeat, https://venturebeat.com/ai/the-risks-of-ai-generated-code-are-real-heres-how-enterprises-can-manage-the-risk/, March 14, 2025.

[27]: Cem Dilmegani, (2025), "Reproducible AI: Why it Matters & How to Improve it [2025]?", https://research.aimultiple.com/reproducible-ai/, February 27, 2025.

[28]: Aatu Väisänen, (2025), "The problem with "vibe coding", https://ikius.com/blog/the-issues-with-vibe-coding, April 24, 2025.

[29]: ARSQA, "Test Environments for AI-Based Systems: ISTQB AI Testing", https://atsqa.org/test-environments-for-ai-based-systems. Retrieved on April 27, 2025.

[30]: KIRUTHIKA R, (2025), "How AI is Rewriting Creativity: From Music to Coding, Are Humans Still Needed?", Dev, https://dev.to/volkmarr/is-generated-code-harder-to-maintain-1n1n, April 23, 2025.

[31]: Poulomi Chakraborty, (2024), "Ultimate Guide to Code Quality and Maintainability in 2024", https://blog.pixelfreestudio.com/ultimate-guide-to-code-quality-and-maintainability-in-2024/. Retrieved on April 26, 2025.

[32] : DORA, "Code maintainability", https://dora.dev/capabilities/code-maintainability/. Retrieved on April 26, 2025.

[33]: D. Coleman, D. Ash, B. Lowther and P. Oman, (1994), "Using metrics to evaluate software system maintainability," in Computer, vol. 27, no. 8, pp. 44-49, Aug. 1994.

[34]: Ukeje Chukwuemeriwo Goodness, (2025), "Maintainability Is All You Need", Dev, https://dev.to/goodylili/maintainability-is-all-you-need-4594, January 9, 2025.

[35]: Stack Exchange, (2019-2020), "Which explainable artificial intelligence techniques are there?", https://ai.stackexchange.com/questions/12870/which-explainable-artificial-intelligence-techniques-are-there.

[36]: Wojciech Samek, Grégoire Montavon, Andrea Vedaldi, Lars Kai Hansen, Klaus-Robert Mü, (2019), "Explainable AI: Interpreting, Explaining and Visualizing Deep Learning", Springer.

[37]: Christoph Molnar, (2021), "Interpretable Machine Learning. A Guide for Making Black Box Models Explainable.", https://leanpub.com/interpretable-machine-learning.