What is Vibe coding and when should you use it (or not)?

Marko Horvat[*]
[*] University of Zagreb, Faculty of Electrical Engineering and Computing
Department of Applied Computing
Unska 3, HR-10000 Zagreb, Croatia
marko.horvat3@fer.unizg.hr

***Abstract* - The invention of Generative AI and Large Language Models has recently given rise to "vibe coding" as a new paradigm of software development in which developers use natural language to state their high-level intent rather than writing or even being aware of the computer code. While the great ease of use of this new paradigm presents outstanding opportunities for Rapid Application Development, it also creates a significant risk of misuse, especially with inexperienced developers as its adoption outpaces formal understanding of the underlying technologies. Currently, there is a lack of empirical research analyzing the fundamental behaviors and code quality of different LLM tools. Moreover, as the software industry is rapidly undergoing profound changes, there is no consensus among practitioners even on the definition of vibe coding.**

This paper provides a formal definition of the vibe coding paradigm, a list of its advantages and disadvantages, a systematic comparation of vibe coding to other AI-assisted programming approaches, and suggestions for the use of this new paradigm.

We argue that vibe coding should be avoided as a primary tool for learning programming, particularly in academic settings, as it can obscure foundational concepts. Furthermore, it is ill-suited for large-scale or production-level projects due to significant debugging and maintenance overhead. Instead, we conclude that it should be used only as a very specific productivity tool for experienced developers engaged in rapid prototyping and experimentation.

Keywords - software engineering; automatic programming; human-computer interaction; HCI; generative AI; Large Language Models; LLMs

I. Introduction

The field of software engineering is currently undergoing a massive paradigm shift, driven by the integration of Generative Artificial Intelligence (GenAI) and Large Language Models (LLMs) into the software development lifecycle [1]. These models are no longer confined to autocompletion but are emerging as proactive virtual partners in application design, code implementation, refactoring, and testing [2]. This evolution has very recently given rise to a term "vibe coding" coined by Andrej Karpathy, which describes a software development methodology centered on a developer's intuitive, natural language expression of intent to an LLM [3]. This practice lowers the entry barrier for code developers, making possible for individuals with minimal formal programming knowledge to create software artifacts. As such, vibe coding signifies a fundamental transition from programming as a formal, syntactic, computer engineering task to a conversational, Human-Computer Interaction (HCI) challenge [4].

Despite the widespread adoption of LLM-based coding assistants [5], because of the fast pace of development there is currently a significant lack of rigorous, comparative studies analyzing their performance and technical characteristics from a software engineering perspective [6] [7]. It remains unclear whether these tools are interchangeable or if they embody different underlying design philosophies that profoundly affect the software development process and the final product [8].

The desire to help those without programming skills is not entirely new and shares many similarities with the established "no-code" paradigm [9][10][11]. Both vibe coding and no-code platforms aim to abstract away the complexity of syntax, allowing users to create applications without writing or necessarily understanding the underlying code. However, they diverge fundamentally in their interaction model and operational principles. No-code development environments, such as historical examples like Apple's HyperCard or Sybase's PowerBuilder, and many others, rely on structured, visual Integrated Development Environments (IDEs) [12]. The user interacts with graphical elements—dragging components, connecting nodes in a workflow, and setting properties in menus. This process is deterministic: a specific set of user actions will reliably and repeatedly produce the same outcome. In stark contrast, vibe coding operates within a purely conversational and linguistic interface. The developer's primary tool is natural language processing (NLP), and the interaction is a dialogue with the LLM. This makes the process inherently probabilistic and non-deterministic; the same prompt can yield different results upon repeated execution, and the final artifact is a product of conversational refinement rather than visual construction.

The remainder of the paper is organized as follows. Section 2 formally defines the vibe coding paradigm. Section 3 discusses vibe coding and AI-assisted programming in the context of HCI. Essential skills required for a successful vibe coder are categorized and listed in Section 4. A detailed examination of who should avoid vibe coding, and who benefits from it are given in Section 5 and 6, respectively. Vibe coding is not completely novel paradigm, as explained in Section 7. Finally, the last section explores the broader implications of the vibe coding trend and concludes with recommendation for future action and research.

II. What is Vibe Coding?

Vibe coding is a novel software development paradigm (as of June 2025) in which a developer expresses their intentions in creating a software product using natural language in a GenAI LLM. Instead of writing precise, line-by-line code, the developer acts as a "high-level coordinator", guiding the AI agent through an iterative dialogic process of repeated code generating prompts and refinements. In vibe coding the key challenge is no longer mastering a programming language and writing software line-by-line but efficiently articulating the desired outcome, i.e., the "vibe", and critically evaluating the results produced by the AI.

It is very important to distinguish vibe coding from any AI-assisted programming because vibe coding does not imply "using AI tools to help write code". Specifically, the term vibe coding implies "**generating code with AI without understanding the code that is produced**". As clearly stated by Andrej Karpathy: "vibe coding is a method of developing throwaway projects that is enjoyable, causing one to forget that the code exists" [3]. This is not the same (in fact, it may be argued it is exactly the opposite) as incorporating LLM tools into a process for the documented and responsible development of production code.

Table I, given below, lists and describes all the key features of vibe coding that differentiate it from AI-assisted coding and traditional programming paradigms. Vibe coding offers an opportunity for most people to develop custom software, even though the majority does not know how or even will not learn to code in a particular programming language or software technology.

III. Vibe Coding and AI-Assisted Programming as End-User Software Engineering in HCI research

Another interesting parallel that can be drawn between vibe coding and AI-assisted programming, and the field of Human-Computer Interaction (HCI), specifically how experienced software engineers and novice programmers interact with software development environments.

For decades, the burden in the software development process was on the human to learn and conform to the machine's rigid, unforgiving syntax. Vibe coding and AI-assisted programming shift this burden, forcing the machine, in the form of an LLM, to interpret the human's ambiguous, high-level intent expressed in a natural language. This transforms software development from a purely formal, syntactic intellectual endeavor into a multifaceted HCI challenge centered on an effective man-machine dialogue.

In this respect, the user of a vibe coding paradigm, the "vibe coder", can be defined as a *novice with minimum prior knowledge of the software development process, or even, person who is not a software developer*. This context positions the vibe coding in the domain of End-User Development (EUD) or End-User Software Engineering (EUSE) [4][13], a HCI field where non-professional developers can create, modify, or extend software artifacts. In this context, LLMs represent a powerful new enabling technology for EUSE, allowing users to specify complex needs in natural language rather than relying on constrained graphical interfaces or simplified scripting languages. LLMs offer a non-deterministic, conversational interface where the quality of the outcome depends entirely on the clarity and iterative refinement of the human-AI dialogue.

TABLE I. Key differences between vibe coding, AI-assisted coding, and traditional programming.

	Vibe Coding	**AI-assisted Coding**	**Traditional Coding**
Developer role	High-level coordinator guiding the AI with natural language dialogue	Coder using AI tools for assistance to augment and accelerate the manual coding process	Only author/developer, who manually writes/implements, structures, and debugs all code line-by-line
Developer core skills	Prompt engineering, manual or assisted evaluation of AI output, iterative refinement of produced code through high-level concepts.	Good programming skills also the ability to effectively use AI tools. Beneficial knowledge of algorithms and data structures, even programming patterns and software architectures.	Deep knowledge of programming language syntax, algorithms, data structures, programming patterns and software architectures.
Code understanding	Not required; the code is a "black box". The paradigm is defined as potentially "generating code with AI without understanding the code that is produced."	Required. The developer owns the code: must understand, review, and take responsibility for all AI-suggested code.	Absolute. The developer is the sole author and must have a complete understanding of the entire codebase.
Interaction method	Conversational dialogue using natural language to express intent (the "vibe").	Writing code in an IDE, with AI providing real-time suggestions, autocompletions, and boilerplate code.	Writing code directly in one or more specific programming languages using an IDE.
Process type	Inherently probabilistic and non-deterministic; the same prompt may not produce the same output.	Hybrid; deterministic (developer's code) + probabilistic (AI suggestions).	Fully deterministic; the same code will always produce the same result.
Intended use	Rapid prototyping and development of non-production projects.	Boosting productivity of professional developers in documented, responsible, production-level software development.	All levels of software development, from small scripts to large-scale production systems.

In this sense, LLMs serve as an enabling technology for a new generation of end-user programming tools. They not only expand the expressiveness available to end-users but also support a form of interaction that is far more aligned with everyday reasoning and communication patterns. This change has the potential to dramatically broaden participation in software creation, but it also raises important concerns regarding accuracy, verification, and the user's ability to understand, debug, or maintain the generated systems. These challenges mirror long-standing issues in HCI, where usability and transparency often trade off against power and flexibility.

Thus, the rise of vibe coding can be seen as both a continuation and a transformation of prior HCI work on making programming more accessible. It prompts a reexamination of fundamental questions in the design of programming systems, including how much abstraction is beneficial, what kinds of scaffolding users need, and how to preserve control and accountability when the software is written by a machine at the user's request.

IV. Essential skills for a Vibe Coder

The essential skills for the new role of vibe coder or vibe programmer are no longer centered on the traditional computer engineering knowledge such as computer science theory, algorithms and methods, object-oriented programming, computer language syntax, programming patterns, or the actual programming language experience, but on the competence of the interaction with the LLM-based chatbot. Specifically, these new skills important for vibe coding include:

Prompt Engineering: The ability to efficiently create precise, specific, and context-rich prompts is paramount to guiding the LLM toward a desired outcome. The methodology employed in this study, which breaks down complex application development into a sequence of iterative prompts, aligns with established prompt engineering best practices like iterative refinement and task decomposition.

Critical Evaluation: The developer must possess the ability to critically assess the quality, correctness, and architectural soundness of the AI-generated code. LLMs frequently introduce subtle errors, security flaws, or inefficient patterns that may function correctly but are poorly designed or may not work at all although LLM is "confident" that the generated code is correct and bug-free.

Iterative Refinement: Crucially, success in vibe coding depends on engaging in a conversational loop of generating code, testing its output, and providing targeted feedback and corrective prompts to the AI to fix deviations and hopefully, progressively converge towards a satisfactory solution. However, this iterative and interactive dialogue does not necessarily always lead to the desired goal, as the LLM can sometimes get "stuck" at a certain point in the development process, showing no further progress despite repeated corrections. In such cases it is often advisable to restart the vibe coding process from the beginning rather than continue an unproductive conversational thread.

V. Who Benefits from Vibe Coding?

Given these special characteristics, it is important to define who the vibe coding is intended for. Although the paradigm seems especially appealing to junior developers, particularly those with minimal formal programming experience, it is equally valuable for senior developers who may lack the time or interest to thoroughly learn every emerging technology or framework. Vibe coding reduces the entry barrier by enabling users to express intent through natural language, rather than requiring them to master new syntax or APIs. This significantly accelerates development, reduces the cognitive load associated with traditional coding practices, and eliminates the "cold start" problem with personal productivity in new technologies by allowing developers to start programming immediately without having to consult manuals and technical documentation or acquiring sufficient experience with a new toolset.

VI. Who Should Not Use Vibe Coding?

However, the simplicity that vibe coding offers can be misleading and potentially harmful particularly for inexperienced programmers. Junior developers often lack the theoretical foundations required to comprehend software architecture, detect hidden errors, and debug ineffectual components. In academic settings, schools and universities, particularly within computer engineering and computer science curricula, relying on vibe coding may cause students to produce suboptimal but sufficiently functional code without understanding the underlying logic, control flow, or data structures. Because vibe coding is inherently simple to use, students might disregard fundamental learning processes, acquiring the ability to prompt for solutions but lacking the underlying necessary analytical foundations that formal programming education provides.

Senior developers, on the other hand, who have already mastered these fundamental skills and learned programming in a traditional, structured way, are better equipped to use vibe coding productively. For them, it is a tool for rapid prototyping, exploring new technologies, and reducing boilerplate overhead. Nonetheless, as the codebase generated by LLMs grows, experienced developers may find it increasingly difficult to trace, understand, and debug such projects. The maintenance challenges of large AI-generated projects could become a significant overhead. The non-deterministic and verbose nature of LLM output may worsen the complexity, introducing inefficiencies and code maintainability issues in the long run.

Therefore, a clear recommendation emerges; **vibe coding should be used as a productivity tool for experienced developers for rapid prototyping or exploring new domains**. **Vibe coding should not be used as a primary method for programming education, nor is it appropriate for developing large-scale or production-grade software systems.**

Table 2 below systematically presents key risks, or dangers, with the adoption of vibe coding for both junior developers, or students at schools and universities, and senior programmers.

TABLE II. KEY DANGERS OF VIBE CODING FOR STUDENTS AND PROFESSIONAL DEVELOPERS WITH DIFFERENT EXPERIENCE LEVELS.

Dangers for students and junior developers	Dangers for senior developers
Obscuring foundational concepts: Vibe coding allows students to create functional applications without understanding the underlying logic, data structures, or software architecture. This creates an “illusion of simplicity” that hinders the development of essential analytical and debugging skills.	**Accumulation of hidden technical debt:** Proactive AI agents can make idiosyncratic or characteristic but poor architectural decisions. An experienced developer might accept a functional prototype without realizing that it has an unscalable or unmaintainable foundation.
Inability to critically evaluate code quality: Lacking experience, students and inexperienced developers cannot distinguish between a well-designed solution and a visually appealing but faulty solution. They assess quality based on superficial functionality, not algorithms, data structures, functions or code reuse, patterns and maintainability.	**Increased maintenance and debugging overhead:** The verbose and often non-standardized code generated by LLMs makes it difficult to track, understand, and maintain the codebase as the project grows. The non-deterministic nature of AI makes it difficult to reproduce and systematically fix bugs.
Dependency on development tools, false sense of competence, shallow understanding, and acquiring poor habits: Relying too much on GenAI tools for code generation can lead to a dependency mindset where students learn to prompt for answers instead of solving problems. This bypasses the crucial learning process of confronting and overcoming coding challenges. What is even worse, students do not realize that there is more to learn and understand.	**Unpredictability and lack of reproducibility:** The probabilistic nature of LLMs means that one and the same prompt can lead to different results. This makes it difficult to establish reliable, repeatable development processes, which are essential for professional software engineering.
Ineffective debugging and refinement: If the generated code fails or needs to be changed, e.g., a failed refactoring attempt, students lack the basic knowledge to debug the "black box" code, leading to frustration and abandonment of the project.	**Risk of inefficient or “buggy” implementations:** While an experienced developer can spot obvious bugs, subtle inefficiencies or security vulnerabilities introduced by the LLM can go unnoticed in large blocks of generated code, and lead to performance or security issues.

Figure 1 presents a decision flowchart for determining the suitability of vibe coding, as a practical "guide" for developers with different levels of experience.

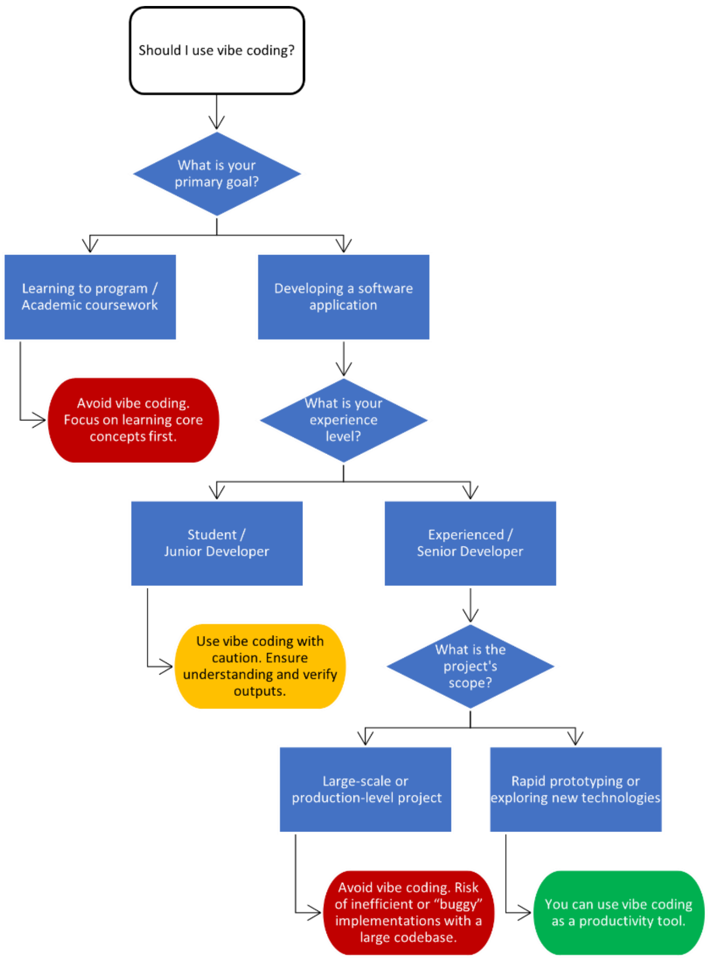
Should I use vibe coding?
What is your primary goal?
Learning to program / Academic coursework
Developing a software application
Avoid vibe coding. Focus on learning core concepts first.
What is your experience level?
Student / Junior Developer
Experienced / Senior Developer
Use vibe coding with caution. Ensure understanding and verify outputs.
What is the project's scope?
Large-scale or production-level project
Rapid prototyping or exploring new technologies
Avoid vibe coding. Risk of inefficient or “buggy” implementations with a large codebase.
You can use vibe coding as a productivity tool.

Figure 1. Decision flowchart for the application of vibe coding in educational and professional software development settings.

VII. Vibe Coding is not completely novel: The "Zero-Code" Analogy

The ambition to give software developers with little to no formal training an ability to create functional applications is not a recent phenomenon born from the GenAI LLM revolution. The core promise of "vibe coding" - enabling creation through high-level intent rather than low-level syntax - actually follows a long history of "zero-code" and Rapid Application Development (RAD) tools that began in the late 1980s [14].

Pioneering examples include Apple's HyperCard, a hypermedia system released in 1987 that allowed users to build interactive programs, presentations, and databases using a simple, but revolutionary at the time, "stack of cards" metaphor [15]. In the 1990s, the client-server era saw the rise of powerful RAD tools that can be now classified as "no-code" or "low-code" [9][10][11][12]. Sybase PowerBuilder, first released in 1991, became famous for its "DataWindow" technology, which enabled developers to easily create and reuse complex, data-driven forms, and reports with minimal coding [16]. Similarly, Borland Delphi, launched in 1995, provided a visual design environment and a component-based architecture that dramatically accelerated the applications development. Other early tools already offered intuitive graphical interfaces for database creation and management, further lowering the barrier to entry for application development [17].

However, a crucial distinction separates these historical precursors from modern vibe coding: the past tools were deterministic systems. Their behavior was predictable; a given set of inputs and actions would always and reliably produce the same output. In contrast, LLM-based vibe coding is inherently probabilistic [18]. The output from an AI prompt is not guaranteed to be identical across repeated attempts, and the models themselves are often "black boxes". Furthermore, just as their predecessors often struggled when faced with requirements for complex or unusual functionality, LLMs can also fail to produce viable solutions for novel or highly specialized tasks [19].

Vibe coding, therefore, can be seen not as a complete breakthrough, but the latest incarnation in a multi-decade quest to make software development simpler and more accessible, is distinguished from AI-assisted programming tools and traditional programming paradigm by several distinct features, as listed in Table 1, but the most by its natural language conversational interface and non-deterministic and stochastic behavior.

VIII. Conlusion

In the academic setting, vibe coding has the potential to transform the educational process. Its integration into university-level programming instruction has an opportunity to increase engagement, accelerate learning, and improve overall teaching quality [20], especially when integrated into one of the many digital game-based learning models that support formal student assessment [21]. However, the increasing ease of automatic code generation also introduces new pedagogical challenges, particularly in academic integrity, plagiarism detection, and assessment design [22] [23]. GenAI tools can help students with coding assignments by suggesting, hinting, and generating code snippets, encouraging better coding practices. However, using GenAI in programming education can lead to students becoming too dependent on AI-generated code and failing to understand fundamental concepts, which may negatively impact the long-term sustainability of computer engineering in higher education [24].

Developing robust solutions for detecting vibe-coded laboratory assignments, seminar papers, and student projects is an urgent issue that the academic community must address [25]. From an application design standpoint, vibe coding encourages the creation of responsive, mobile-friendly, and platform-consistent interfaces, allowing students and professionals to create modern frontends with minimum technical overhead [26].

In the future, it will be necessary to develop clear guidelines, standards, and pedagogical frameworks for the responsible use of vibe coding in education and the workplace. Educational institutions such as schools and universities should consider introducing hybrid teaching models in which vibe coding is used as a complementary tool and not as a substitute for traditional programming instruction. Such hybrid models would preserve the cultivation of fundamental algorithmic thinking while capitalizing on the productivity gains offered by GenAI. In addition, new assessment methods should be developed, potentially combining behavioral analysis, interactive code reviews, live or oral exams, and traceable revision histories to accurately assess student learning outcomes in environments where AI support is ubiquitous.

For professional developers, vibe coding opens new avenues for rapid ideation, cross-domain prototyping, and on-demand translation of human intent into code, but it requires rigorous quality controls to ensure maintainability, reproducibility, and security. As LLMs become more prevalent in development, automated tools for verification, refactoring and standardization of AI results are becoming increasingly important components of the software engineering ecosystem.

References

[1] Coello, C. E. A., Alimam, M. N., & Kouatly, R. (2024). Effectiveness of chatgpt in coding: A comparative analysis of popular large language models. *Digital*, *4*(1), 114-125.

[2] Kotsiantis, S., Verykios, V., & Tzagarakis, M. (2024). AI-assisted programming tasks using code embeddings and transformers. *Electronics, 13*(4), 767.

[3] Karpathy, A. (2025, Feb 3). Vibe coding. Retrieved from https://x.com/karpathy/status/1886192184808149383

[4] Gunatilake, H., Grundy, J., Hoda, R., & Mueller, I. (2024). The impact of human aspects on the interactions between software developers and end-users in software engineering: A systematic literature review. *Information and Software Technology*, 107489.

[5] Porter, L., & Zingaro, D. (2024). *Learn AI-Assisted Python Programming: With Github Copilot and ChatGPT*. Simon and Schuster.

[6] Shakya, R., Vadiee, F., & Khalil, M. (2025, April). A Showdown of ChatGPT vs DeepSeek in Solving Programming Tasks. In *2025 International Conference on New Trends in Computing Sciences (ICTCS)* (pp. 413-418). IEEE.

[7] Liang, J. T., Yang, C., & Myers, B. A. (2024, February). A large-scale survey on the usability of ai programming assistants:

Successes and challenges. In *Proceedings of the 46th IEEE/ACM international conference on software engineering* (pp. 1-13).

[8] Sergeyuk, A., Golubev, Y., Bryksin, T., & Ahmed, I. (2025). Using AI-based coding assistants in practice: State of affairs, perceptions, and ways forward. *Information and Software Technology*, *178*, 107610.

[9] Hurlburt, G. F. (2021). Low-code, no-code, what's under the hood?. *IT professional*, *23*(6), 4-7.

[10] Rokis, K., & Kirikova, M. (2022, September). Challenges of low-code/no-code software development: A literature review. In *International conference on business informatics research* (pp. 3-17). Cham: Springer International Publishing.

[11] El Kamouchi, H., Kissi, M., & El Beggar, O. (2023, November). Low-code/No-code Development: A systematic literature review. In *2023 14th International Conference on Intelligent Systems: Theories and Applications (SITA)* (pp. 1-8). IEEE.

[12] Hirschberg, M. A. (1998). Rapid application development (rad): a brief overview. *Software Tech News*, *2*(1), 1-7.

[13] Robinson, D., Cabrera, C., Gordon, A. D., Lawrence, N. D., & Mennen, L. (2025). Requirements are all you need: The final frontier for end-user software engineering. *ACM Transactions on Software Engineering and Methodology*, *34*(5), 1-22.

[14] Beynon-Davies, P., Carne, C., Mackay, H., & Tudhope, D. (1999). Rapid application development (RAD): an empirical review. *European Journal of Information Systems*, *8*(3), 211-223.

[15] Larsen, M. D. (1988). Dealing your own hand with HyperCard. *Hispania*, *71*(2), 451-457.

[16] Anqi, Y., Jinglan, Y., Yaowen, Y., Dujuan, Z., & Xiang, L. (2006, November). The PowerBuilder design technology for the data window print preview and print. In *2006 7th International Conference on Computer-Aided Industrial Design and Conceptual Design* (pp. 1-4). IEEE.

[17] Naz, R., & Khan, M. N. A. (2015). Rapid applications development techniques: A critical review. *International Journal of Software Engineering and Its Applications*, *9*(11), 163-176.

[18] Taulli, T. (2024). *AI-Assisted Programming: Better Planning, Coding, Testing, and Deployment*. " O'Reilly Media, Inc.".

[19] Jing, Y., Wang, H., Chen, X., & Wang, C. (2024). What factors will affect the effectiveness of using ChatGPT to solve programming problems? A quasi-experimental study. *Humanities and Social Sciences Communications*, *11*(1), 1-12.

[20] Šarčević, A., Tomičić, I., Merlin, A., & Horvat, M. (2024, May). Enhancing Programming Education with Open-Source Generative AI Chatbots. In *2024 47th MIPRO ICT and Electronics Convention (MIPRO)* (pp. 2051-2056). IEEE.

[21] Horvat, M., Jagušt, T., Veseli, Z. P., Malnar, K., & Čižmar, Ž. (2022, May). An overview of digital game-based learning development and evaluation models. In *2022 45th Jubilee International Convention on Information, Communication and Electronic Technology (MIPRO)* (pp. 717-722). IEEE.

[22] Hartley, K., Hayak, M., & Ko, U. H. (2024). Artificial intelligence supporting independent student learning: An evaluative case study of ChatGPT and learning to code. *Education Sciences*, *14*(2), 120.

[23] Mekterović, I., Brkić, L., & Horvat, M. (2023). Scaling automated programming assessment systems. *Electronics*, *12*(4), 942.

[24] Silva, C. A. G. D., Ramos, F. N., De Moraes, R. V., & Santos, E. L. D. (2024). ChatGPT: Challenges and benefits in software programming for higher education. *Sustainability*, *16*(3), 1245.

[25] Adnin, R., Pandkar, A., Yao, B., Wang, D., & Das, M. (2025, April). Examining Student and Teacher Perspectives on Undisclosed Use of Generative AI in Academic Work. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems* (pp. 1-17).

[26] Smolić, E., Boras, B., Horvat, M., & Jagušt, T. (2024). Smartphone-Enabled Interaction on Large Displays—A Web-Technology-Based Approach. *Electronics*, *13*(5), 929.