A Comparative Study of Vibe Coding with ChatGPT and Gemini in Front-end Web Development

Marko Horvat, Barbara Kralj, Gordan Gledec
University of Zagreb, Faculty of Electrical Engineering and Computing
Department of Applied Computing
Unska 3, HR-10000 Zagreb, Croatia
{Marko.Horvat3, Barbara.Kralj, Gordan.Gledec}@fer.unizg.hr

Abstract. *The invention of Generative AI and Large Language Models has recently catalyzed "vibe coding" as a new paradigm of software development in which developers use natural language to state their intentions. However, there is currently a significant lack of empirical research comparing the fundamental behaviors of GenAI tools and their code quality. This paper presents such comparative study of GPT-4o and Gemini 2.5 Pro for front-end web development using everyday technologies HTML, CSS, and JavaScript. Using zero-shot and prompt-chaining strategies, we tasked the models to create three commonplace web applications of increasing complexity. The architecture and features of the generated code were evaluated using a mixed-method evaluation framework. The results show that GPT-4o and Gemini 2.5 Pro represent two different development paradigms; GPT-4o functions as a tool that generates minimal, concise code that follows user instructions, but in more complex tasks it sometimes generates errors and unwanted changes in the codebase. In contrast, Gemini 2.5 Pro operates as a proactive-enhancement agent that generates more complex, feature-rich code by anticipating the user's needs and adding advanced UI functionalities. Importantly, vibe coding is formally defined, explained, and compared to other AI-assisted programming approaches. The codebase created for this research is available at: https://github.com/mhorvat/vibecoding_frontend.*

Keywords. Software engineering, Automatic programming, Human-computer interaction, Generative AI, Large Language Models

1 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 (Coello, Alimam & Kouatly, 2024). These models are no longer confined to autocompletion but are emerging as proactive virtual partners in application design, code implementation, refactoring, and testing (Kotsiantis, Verykios & Tzagarakis, 2024). 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 (Karpathy, 2025). 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 (Gunatilake et al., 2024).

Despite the widespread adoption of LLM-based coding assistants (Porter & Zingaro, 2024), 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 (Shakya, Vadiee & Khalil, 2025) (Liang, Yang, & Myers, 2024). 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 (Sergeyuk et al., 2025).

The presented research addresses this gap by evaluating the two most commonly used general-purpose LLM models, OpenAI's GPT-4o and Google's Gemini 2.5 Pro, on vibe coding of three typical front-end development tasks of increasing complexity. The trade-offs in vibe coding between the two models are analyzed and their implications discussed for novice developers, who are increasingly turning to these tools as their primary means of learning and building software, as well as senior developers who may turn to vibe coding for quick prototype building or assessment of new software technologies.

The remainder of the paper is organized as follows: the next section defines the vibe coding paradigm and explains its advantages and disadvantages in the software development process, the third section details the experimental methodology, while the fourth section

presents the experimental results. The fifth section discusses the broader implications of our findings. Finally, the last section concludes with directions for future research and recommendations for academia.

2 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" (Sapkota, Roumeliotis & Karkee, 2025). Specifically, the term vibe coding can be defined as "**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" (Karpathy, 2025). 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 1 lists and briefly describes the key features of vibe coding that differentiate it from AI-assisted coding and traditional programming paradigms.

Table 1. Key differences between vibe coding, AI-assisted coding, and traditional programming

	Vibe Coding	AI-assisted Coding	Traditional programming
Developer role	Coordinator guiding the AI with natural language	Coder using AI tools for assistance	Only author, writing all code manually
Developer core skills	Prompting, evaluation, and iterative refinement	Programming skills and AI tool proficiency	Mastery of syntax, algorithms, and architecture
Code understanding	Not required; the code is a "black box"	Required; the developer owns all code	Absolute; the developer is the author
Interaction method	Natural language dialogue	IDE with AI autocompletion and suggestions	Directly writing code in an IDE
Process type	Probabilistic and non-deterministic	Hybrid (deterministic + probabilistic)	Fully deterministic
Intended use	Rapid prototyping and non-production projects	Productivity boost for professional developers	All development, including production systems

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.

2.1 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.

2.2 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.

2.3 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 ultimate 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.

2.4 Vibe Coding as End-User Software Engineering in HCI research

The user of a vibe coding paradigm, i.e., **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) (Robinson et al., 2025) (Gunatilake et al., 2024), a HCI field where non-professional developers 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.

2.5 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 (Beynon-Davies et al., 1999).

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 "stack of cards" metaphor. In the 1990s, the client-server era saw the rise of powerful RAD tools (Hirschberg, 1998). Sybase PowerBuilder, first released in 1991, became famous for its "DataWindow" technology, which enabled developers to create complex, data-driven forms, and reports with minimal coding. 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 (Naz & Khan, 2015).

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 (Taulli, 2024). 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 (Jing et al., 2024).

Vibe coding, therefore, can be seen not as a complete breakthrough. It is only the latest incarnation in a multi-decade quest to make software development simpler and more accessible. Vibe coding 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.

3 Experimental Setup

The objective of the conducted experiment was to empirically compare the performance, generated software components, and interactive behavior of OpenAI's GPT-4o and Google's Gemini 2.5 Pro models as would be used by a novice developer for frontend web development tasks.

The experiment was intentionally constrained to foundational web technologies to isolate the LLMs' core generative capabilities without the potentially many confounding variables of newer, complex, or less popular frameworks which might have less represented in the LLMs' training set. The chosen stack included:

- HTML: For web document structure.
- CSS with Bootstrap: For frontend styling and responsive layout.
- Plain ("vanilla") JavaScript: For client-side dynamic logic, deliberately avoiding frameworks like React, Node.js or Vue.js to better assess the models' basic coding abilities.

We tasked the models to create three characteristic web frontends of increasing complexity: 1) simple paraphrasing and spellchecking tool, 2) moderately complex messenger client, and 3) complex e-commerce web shop. The first two applications required only basic HTML, CSS, and JavaScript without external graphical assets, and we provided the models with a predefined set of images and icons to be incorporated into the web shop interface. Additionally, a reference picture was provided to the models as a visual template to illustrate the intended appearance of the web shop frontend layout.

A zero-shot prompt chaining approach was used to instruct the models through the code development process. Specifically, for each application, a predefined set of sequential prompts defining web features and frontend layout was given to each model. After each prompt, the generated code was analyzed, and corrective follow-up prompts were issued as needed to fix errors or address misinterpretations. This process simulates a realistic, conversational development workflow in the vibe coding paradigm. All interactions were conducted in English, the *de facto* language for software development and the primary language for LLM training data. In the final step of the experiment, the generated code for the web shop was exchanged between the two models, with each model tasked with refactoring the other's output. A long and detailed zero-shot prompt was used for the final refactoring.

The sequence of prompts for each interface, the paraphrasing tool, the messenger client, and the web shop, is shown in Fig. 1. A custom metrics was defined containing a comprehensive set of various vibe coding features to evaluate each model's performance. The metrics is explained in Table 2.

Table 2. Model evaluation metrics used in the experiment.

Category	Feature	Description
Interaction effort	Number of instructional prompts	The number of initial prompts required to define a specific frontend task.
	Number of corrective prompts	The number of follow-up prompts needed to fix errors or deviations.
	Total number of prompts	The sum of all interactions with the model.
Code quality and size	Total Lines of Code (LOC)	The total volume of generated HTML, CSS, JS code (excluding comments).
	Number of unrequested features	Functionalities generated by the LLM that were not explicitly asked for in the prompts. Model proactivity.
Reliability	Number of errors	The number of functional or syntactic errors in the generated code.
	Number of misinterpreted prompts	Instances where the model significantly misunderstood a core requirement.
User perception	Subjective rating (1-5)	User rating of the final solution's quality, functionality, and adherence to the initial "vibe."

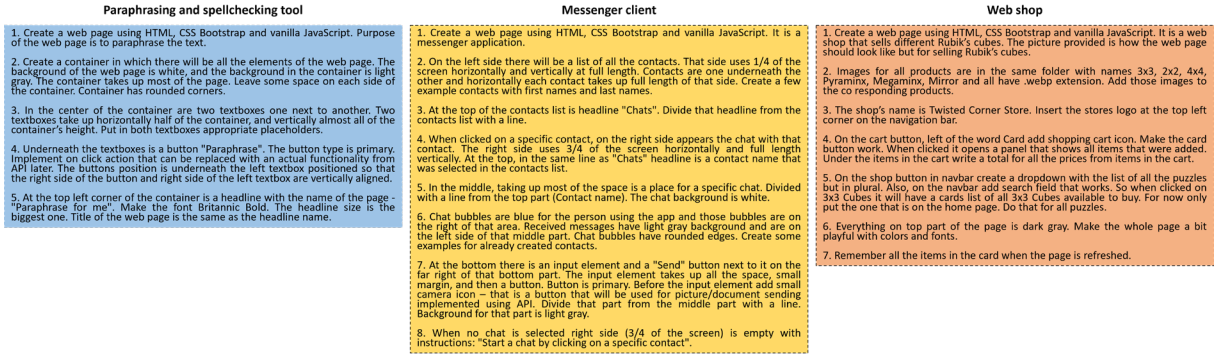

Figure 1. The sequence of prompts used to vibe code the three web frontends (paraphrasing tool, messenger client and web shop) of different complexity, the simplest on the left, and the most complex on the right.

4 Results and analysis

The final appearance of the paraphrasing frontends for both evaluated models is shown in Fig. 2.

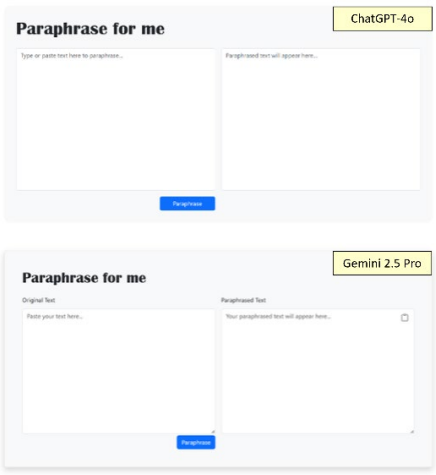

Figure 2. Comparison of paraphrasing tool frontends generated by GPT-4o (top) and Gemini 2.5 Pro (bottom).

As can be seen in Fig. 2, both versions share a similar minimalist design but there are notable differences in usability and visual clarity. Gemini's interface includes explicit field labels ("Original Text" and "Paraphrased Text"), which improve clarity and usability. Also, it incorporates a copy-to-clipboard button in the output field, providing a functional enhancement missing in the GPT-4o code. Overall, although similar Gemini's layout provides a more structured visual design with slightly better user experience (UX) and web interface quality.

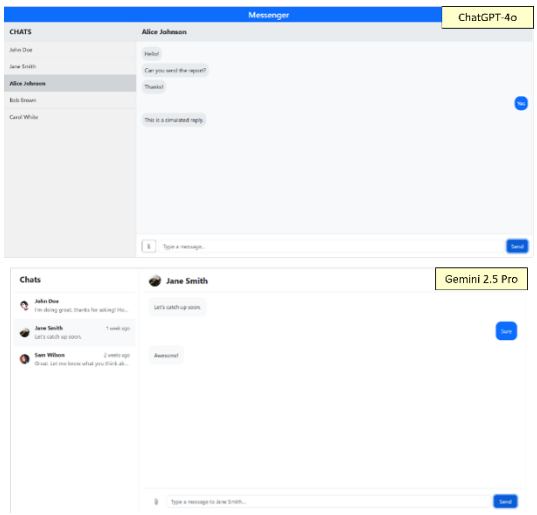

Figure 3. Comparison of messenger tool frontends generated by GPT-4o (top) and Gemini 2.5 Pro (bottom).

As with the first frontend task, in the second task Gemini 2.5 Pro's messenger frontend shows more UX maturity by including profile images, timestamped messages, and chat previews in the sidebar (Fig. 3). These features improve user orientation and make the interface feel more like a modern messaging platform. The GPT-4o version, on the other hand, offers a minimalist layout with functional but less engaging visual elements, without avatars, time context and preview content. Although both designs include quick reply buttons, and functional input areas, Gemini's user interface is richer, more intuitive to modern users.

In the last task, generating a web shop, the models were given a prompt and an image with a template for the layout of the user interface (see Fig. 4).

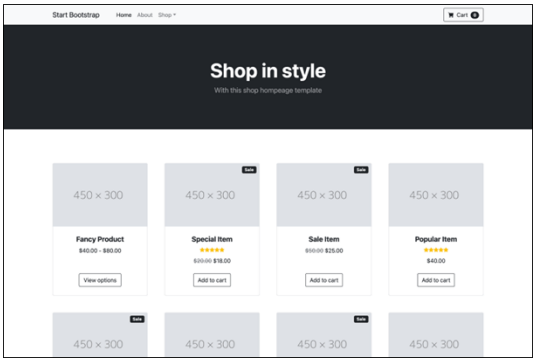

Figure 4. The webshop frontend layout template given to the models together with the prompts.

The output of the webshop vide coding task is presented in two parts. In addition to generating the homepage of the webshop homepage, both models also produced a shopping cart layout. The main layout of the front end is shown in Fig. 5, while the shopping cart is presented separately in Fig. 6.

The webshop frontend generated by Gemini appears more complete. It contains additional features such as product discount display, a sales label, a structured footer, and a more extensive product catalogue. The GPT-4o model provides a more colorful and well-rounded aesthetic with simpler product labelling and fewer UI enhancements. The Gemini vibe code for the shopping cart interface has a higher level of design maturity and functional completeness compared to GPT-4o. Gemini includes important usability improvements such as product thumbnail images, clearly defined quantity indicators, intuitive delete icons, and a prominently placed checkout button that allows for better e-commerce experience. The GPT-4o is functionally appropriate and sufficient, but lacks visual context and transactional cues, making it more suitable for early prototyping than for a user-ready deployment.

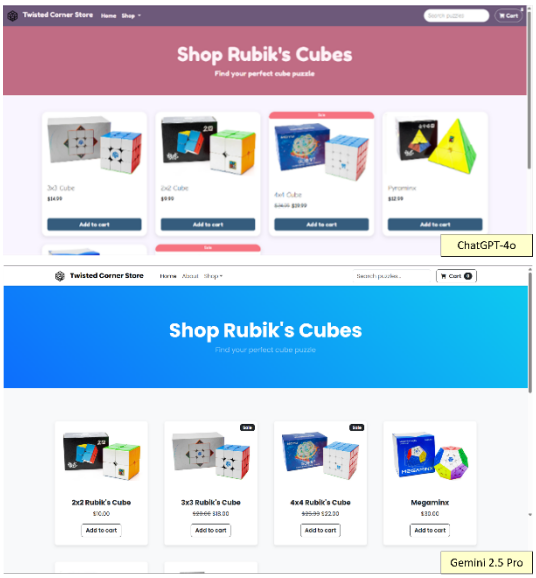

Figure 5. Comparison of webshop frontends created by GPT-4o (top) and Gemini 2.5 Pro (bottom).

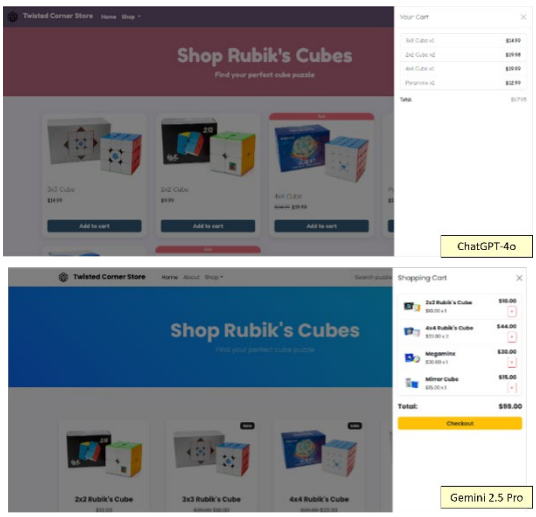

Figure 6. The shopping cart layout in webshop frontends from GPT-4o (top) and Gemini 2.5 Pro (bottom).

Regarding the webshop task, similarly to previous tasks, it can be concluded that while GPT-4o version is functional and visually appealing for a prototype, Gemini's design is more in line with modern e-commerce standards, and supports scalability, visual elements hierarchy, and distinctive marketing elements.

The analysis of the generated code revealed that the aggregated quantitative results of all three applications exhibited consistent variations in the behavior of the employed models. First, Gemini 2.5 Pro consistently produces significantly more code than GPT-4o. This gap becomes larger as task complexity increases, from a 61% larger codebase for the simple paraphrasing tool to an 89% larger one for the most complex webshop. Second, Gemini consistently generates unrequested but useful UI features (e.g., loading spinners, checkout buttons), whereas GPT-4o does not generate extra features and strictly follows the user's prompts. Third, GPT-4o makes errors that require re-entering prompts that have already been entered in the prompt sequence, sometimes several steps before. Fourth, Gemini consistently received higher subjective satisfaction ratings from the standpoint of a novice user. Finally, Gemini's proactive nature sometimes allows it to anticipate future steps, reducing the number of instructional prompts needed, as seen in the Messenger frontend task. The aggregated quantitative results are shown in Table 3.

Table 3. Quantitative performance metrics results.

Metric	Task	ChatGPT	Gemini
Number of instructional prompts	Paraphrase	5	5
	Messenger	7	4
	Web Shop	7	7
Total number of prompts	Paraphrase	14	10
	Messenger	21	13
	Web Shop	14	17
Number of corrective prompts	Paraphrase	9	5
	Messenger	14	9
	Web Shop	7	10
Total Lines of Code (LoC)	Paraphrase	84	135 (+61%)
	Messenger	120	147 (+23%)
	Web Shop	193	365 (+89%)
Unrequested Features	Paraphrase	0	2
	Messenger	0	2
	Web Shop	0	4
Avg. Subjective Rating (1-5)	Paraphrase	3.6	3.8
	Messenger	4.0	4.5
	Web Shop	3.7	4.1

Interestingly, there were no syntax errors in code from either model, in either language – HTML, CSS or JavaScript. The total number of misinterpreted prompts was 1 for GPT-4o, and 0 for Gemini 2.5 Pro. The subjective rating of the quality of the produced final output is given in Table 1.

A qualitative analysis of the code confirms already noticed differences in the architectural and stylistic decisions of each model. While Gemini's proactive approach resulted in a better and more dynamic architecture for the medium-complexity Messenger task, the same tendency proved negative in the high-complexity case webshop. Gemini 2.5 Pro hardcoded a large number of product entries directly into HTML, creating a rigid and unscalable codebase. In contrast, GPT-4o model used a more robust and traditional software engineering pattern: separating data (a JavaScript array of products) from presentation (dynamically rendered HTML). This approach is by design more manageable and scalable.

Because of the space constraints it is not possible to present the entire generated code in full detail; however, several representative elements can be highlighted.

For example, in the first scenario for the paraphrasing tool, the overall code quality produced by both agents is high. The generated code includes all requested components, is well-structured, and uses intuitive variable naming. GPT-4o implementation relies on a single main container (`paraphrase-container`) (Fig. 7) for element layout, whereas Gemini 2.5 Pro introduces additional components such as card and box-shadow, giving the application a more professional appearance (Fig. 7).

GPT-4o

```
<div class="paraphrase-container">
    <h1>Paraphrase for me</h1>
    <div class="content-row">
        ...
    </div>
</div>
```

Gemini Pro 2.5

```
<style>
.card {
    width: 100%;
    height: 100%;
    background-color: #f8f9fa; /* Light gray background */
    border: 1px solid #dee2e6;
    box-shadow: 0 0.5rem 1rem rgba(0, 0, 0, 0.15) ;
    display: flex; /* Use flexbox for the card itself */
    flex-direction: column; /* Stack card content vertically */
}
</style>
...
<div class="content-wrapper">
    <div class="card">
        <div class="card-body p-4 p-md-5">
            <h1 class="card-title mb-4">Paraphrase for me</h1>
            <div class="grid-container">
                ...
            </div>
        </div>
    </div>
</div>
```

Figure 7. The main HTML container for element layout in paraphrasing tool generated by GPT-4o (above) and Gemini 2.5 Pro (below).

In JavaScript, GPT-4o provides a minimal implementation with only the basic `onClick` action (Fig. 8), while Gemini delivers a more extensive and detailed implementation (Fig. 8). Notably, Gemini disables the "Paraphrase" button during text processing and adds a loading spinner, further improving usability. Finally, the layout in the ChatGPT-generated interface is static, whereas Gemini's interface exhibits greater flexibility and adaptability. A similar pattern emerges in the Messenger client task and the web shop task: GPT-4o versions are minimal, displaying only the necessary, while Gemini enriched the implementation with additional usability features improving both aesthetics and functionality.

The final step of the experiment was a refactoring scenario with the webshop code previously generated by the two agents. Specifically, the ChatGPT agent was assigned the code originally generated by the Gemini agent and, conversely, the Gemini agent received the code generated by ChatGPT. Along with the code, both agents were provided with a clearly defined prompt outlining how to approach the refactoring task. Upon receiving the instructions, both agents identified issues in the code and presented concise summaries of the problems, followed by detailed implementation steps for the refactoring process and best practice recommendations for future development.

GPT-4o

```
document.getElementById('paraphraseBtn').addEventListener('click', () => {
    const inputText = document.getElementById('inputText').value.trim();
    const outputTextarea = document.getElementById('outputText');
    outputTextarea.value = '';
    if (!inputText) {
        outputTextarea.value = 'Please enter some text to paraphrase.';
        return;
    }
    // Placeholder action : replace with actual API call
    console.log('Paraphrase button clicked. Input:', inputText);
    outputTextarea.value = '...paraphrased text will appear here...';
});
```

Gemini Pro 2.5

```
const paraphraseBtn = document.getElementById('paraphrase-btn');
const inputText = document.getElementById('inputText');
const outputText = document.getElementById('outputText');
const spinner = paraphraseBtn.querySelector('.spinner-border');
const copyBtn = document.getElementById('copy-btn');
paraphraseBtn.addEventListener('click', () => {
    ...
    spinner.classList.remove('d-none');
    ...
});
copyBtn.addEventListener ('click', () => {
    ...
});
...
```

Figure 8. JavaScript implementation of the button click handler in GPT-4o (above) and Gemini 2.5 Pro (below).

In the refactoring both agents reorganized the original code base into multiple files and directories to improve readability and make it easier to find components, as shown in Fig. 9. All business logic was moved to JavaScript files: 5 different files in GPT-4o refactoring, and only one JavaScript file (`main.js`) with Gemini 2.5 Pro. In addition, both agents introduced standardized conventions for naming components, adhering to the BEM (Block Element Modifier) method.

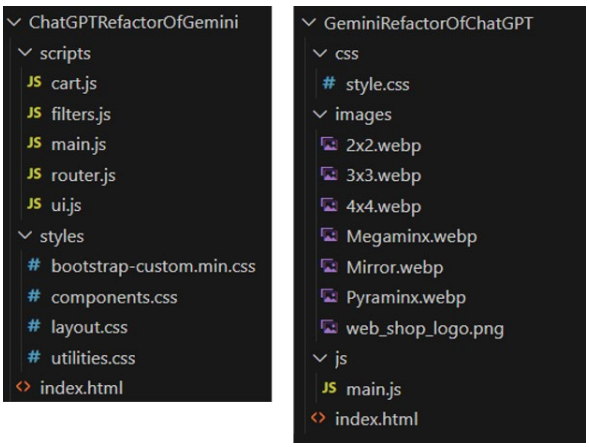

Figure 9. The file structure of the refactored code for the web shop task from GPT-4o (left) and Gemini 2.5 Pro (right).

However, despite providing informative comments and useful suggestions, the code refactored by GPT-4o performed worse than the original implementations. Moreover, the webshop refactored by GPT-4o was not functional because important elements such as the product display and the top navigation bar were

missing. The GPT-4o model assumed that the products would be located in a data directory within a `products.json` file, but failed to specify this requirement or provide an example of the expected JSON structure.

On the other hand, Gemini 2.5 Pro successfully refactored the code from GPT-4o and created an almost identical version of the original page. Despite minor shortcomings, such as the placement of the "Sale" button in the top right corner and oversized product images, the Gemini agent performed a more effective refactoring than GPT-4o. It preserved all the original functionalities and introduced subtle improvements.

The complete codebase developed for this research, consisting of HTML, JavaScript, and CSS files for all four tasks or scenarios (paraphrasing, messenger, webshop and refactoring), is freely accessible at: *https://github.com/mhorvat/vibecoding_frontend*.

5 Discussion

The key finding of the conducted experiment in front-end web development is that GPT-4o and Gemini 2.5 Pro are not simply interchangeable tools for vibe coding. They embody two different philosophies of AI-powered support that represent different levels of support for a developer.

OpenAI's ChatGPT with GPT-4o model acts as a direct and literal tool. Its behavior is predictable and controllable. It produces lean code that strictly adheres to the user's explicit instructions. This paradigm places the entire cognitive burden of design, functional specification, and architectural planning on the user.

Google Gemini 2.5 Pro works as a proactive and collaborative agent. It attempts to derive higher-level goals, anticipate the user's needs, and enrich the output with features and subtleties that go beyond specifications stated in the prompt. It shares the burden of design and makes decisions to deliver a more complete product.

For a junior developer, the choice between these two paradigms is crucial, and our results show that Gemini's approach is generally superior. The data show that Gemini consistently achieves higher subjective satisfaction ratings across all three front-end scenarios. This is because its proactive enhancements provide a more complete and professional looking application that is ready to use immediately. For a beginner, it is very motivating to see how quickly an application with many functions can be created.

The literalness of GPT-4o leads to simpler code, but it is also a significant disadvantage for junior users. Beginner developers must behave like experts, specify every design detail and anticipate web front-end features they may not even know are possible. This may be difficult for junior users and can lead to a frustrating development experience. Gemini's ability to "fill in the gaps" in the prompt, i.e., the software specification, makes it a far more effective assistant, guiding the novice to a better result.

However, Gemini's proactive nature, while beneficial, is not without risks. The "complexity-architecture inversion" observed in the webshop task, where Gemini's hard-coded HTML was architecturally inferior to dynamic data array generated by GPT-4o is a clear example of this feature. But for a developer who is more interested in rapid prototyping rather than building a large-scale enterprise-level application this is a worthwhile trade-off.

The primary goal of any developer using vibe coding should be to quickly turn an idea into a working software prototype. In this context, Gemini's delivery of a feature-rich, visually polished application, and more rapid development and validation is more valuable than the architectural simplicity of GPT-4o code.

Finally, the presented study has several limitations that must be acknowledged. First, the experiment did not include many other popular coding AI tools. An expanded experiment including even more LLMs, and frontend types is planned for future research and a more comprehensive publication. Second, the experiment was conducted with ratings from a single user which limits the generalizability of the obtained results. A formal verification should be undertaken in the future to assess the quality of the generated code more clearly. Third, the study was limited to vanilla HTML, CSS, and JavaScript. The observed paradigms may differ significantly for more complex frameworks such as Node.js, Vue.js or React. Finally, the experiment was focused solely on front-end UI development. The reported results may not apply to other application architecture components such as backend logic, database management or middleware, and should be investigated separately in the future.

6 Conclusion

The presented research provides a direct empirical comparison of OpenAI GPT-4o and Google Gemini 2.5 Pro LLMs in a vibe coding context of web frontend, showing that for junior and senior developers alike, the models are not only different, but one is demonstrably superior. Google Gemini acts as a proactive and augmenting agent, proving to be the more effective tool for both user groups. Its ability to anticipate user requirements, incorporate beneficial unsolicited features, and create more refined and comprehensive web frontend components is perfectly aligned with the goals of an inexperienced developer looking to convert a high-level "vibe" quickly and efficiently into a functional product. Also, for senior developers a proactive model such as Gemini 2.5 Pro will be more beneficial to develop "quick and dirty" code, but with a caveat of a larger codebase than with using GPT-4o. Such larger code will be more difficult to trace and

debug. However, this is a trade-off that a senior developer must contend with.

In the academic setting, vibe coding certainly 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 (Šarčević et al., 2024), especially when integrated into one of the many digital game-based learning models that support formal student assessment (Horvat et al., 2022). However, the increasing ease of automatic code generation also introduces new pedagogical challenges, particularly in academic integrity, plagiarism detection, and assessment design (Hartley, Hayak & Ko, 2024) (Mekterović, Brkić & Horvat, 2023). 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 (Silva, 2024).

Developing robust solutions for detecting vibe-coded laboratory assignments, seminar papers, and student projects is an urgent issue that the academic community must address (Adnin et al., 2025). 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 (Smolić et al., 2024).

In addition to limitations of the experiment, as outlined in the previous section, which should be corrected with future investigations, this research opens several interesting avenues for further research. First, including other models such as Anthropic Claude[1], GitHub Copilot[2] or Cursor AI[3]. Second, by expanding the scope by replicating the study with more complex JavaScript frameworks and software technologies to determine if the observed paradigms remain consistent across the technology stack. Third, by using professional tools and formal code analysis techniques to determine the quality of the generated code. Fourth, the robustness of the vibe coding paradigm should be tested on different programming assignments, such as backend, databases, microservices, etc. Finally, in the future it would be interesting to investigate the impact of different prompting strategies and measure the impact of prompt structure on code quality, architectural choices, and model behavior more systematically.

Universities and curriculum designers must formulate explicit standards for the ethical and successful integration of vibe coding into coursework, ensuring that AI-assisted coding and vibe coding enhance rather than undermine the development of fundamental programming skills.

We hope our findings will have significant positive implications for tool selection and software developer education in vibe coding.

References

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).

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.

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.

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.

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.

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

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.

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.

Karpathy, A. (2025, Feb 3). Vibe coding. Retrieved from

[1] https://www.anthropic.com/claude.
[2] https://github.com/copilot
[3] https://cursor.com/

https://x.com/karpathy/status/1886192184808149383

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

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).

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

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.

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

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.

Sapkota, R., Roumeliotis, K. I., & Karkee, M. (2025). Vibe coding vs. agentic coding: Fundamentals and practical implications of agentic ai. *arXiv preprint arXiv:2505.19443*.

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.

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.

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.

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

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.

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