Proceedings of the 3rd International Conference on Mathematical Physics and Computational Simulation
DOI: 10.54254/2753-8818/100/2025.21617

LLM-Based Web Generation Quality Assessment

Yizhen Gong[1,a,*] , Xiaoyan Wang[1,b]

[1]*Lanzhou University, Chengguan District, Lanzhou City, Gansu Province, China*
a. 320220941540@lzu.edu.cn, b. wangxiaoy@lzu.edu.cn
**corresponding author*

Abstract: Large Language Models (LLMs) have demonstrated powerful capabilities in the field of code generation, with a deep understanding of the semantics and functionality of code. Building websites is one of the most important tasks in software development, as it utilizes rich frontend displays and backend processing to achieve various service functions. It is one of the most widely used interactive software models. Although there have been some efforts in Web website generation, these efforts have been limited to the automation of generating Web pages. The advent of LLMs provides a new approach to Web site generation tasks. However, there is currently a lack of comprehensive evaluation of the generation performance of LLMs in this context, making it difficult to optimize and improve the generated results in a targeted manner. To address this issue, this paper conducts a multi-angle investigation and analysis of the performance of LLMs in Web site generation tasks. Firstly, Web generation requirements are collected, and effective prompt engineering is designed. These prompts are then input into different LLMs to initiate the self-iteration process. Next, the generated code is fed back into the LLM for security self-iteration, where the model performs vulnerability detection and repair on the code it has generated. The security-enhanced code is subsequently subjected to manual review, where it is evaluated using predefined quantitative metrics to generate indicator values. Finally, through testing, the quality, security, and code defects of the generated front-end and back-end Web code across different LLMs are analyzed, providing a comprehensive evaluation of the generation results. The experiments demonstrate that LLM systems perform well in completing and implementing the functions and layouts of pages in prompts for Web generation tasks, but there remains room for improvement in the security of the Web code.

Keywords: LLM, code generation, Web security, defect detection, software development

1. Introduction

With the rapid development of artificial intelligence technologies, Large Language Models (LLMs) have demonstrated powerful capabilities in the field of code generation [1]. These models not only generate high-quality code but also have a profound understanding of the semantics and functionality of code, offering new possibilities for software development. Particularly in the field of Web development, the application potential of LLMs is especially significant. Building websites is one of the key tasks in software development, with its core focus on achieving diverse service functions through rich frontend displays and backend processing technologies. As the most widely used interactive software model, Web development encompasses a broad range of scenarios, from simple

DOI: 10.54254/2753-8818/100/2025.21617

static pages to complex dynamic applications. However, despite the relative maturity of Web development technologies, its development process still faces challenges in terms of efficiency, quality, and security.

In recent years, several Web website generation tools based on automation technologies have emerged, including Web automation testing tools [2] and Web development automation tools [3] . However, these tools are often limited to specific functions or frameworks, making it difficult to meet the demands of complex scenarios. For example, existing tools may only support the generation of static pages or rely on predefined templates, lacking flexibility and scalability. Additionally, these tools have certain limitations in terms of code quality, security, and maintainability. With the rise of LLM technologies, Web site generation tasks are presented with new opportunities. Leveraging their powerful semantic understanding and generation abilities, LLMs can provide more intelligent solutions for Web development. However, despite the significant advantages shown by LLMs in the field of code generation, their actual performance in Web generation tasks has yet to be systematically evaluated and optimized.

Currently, there is a lack of comprehensive research and analysis on the performance of LLMs in Web generation tasks. Existing research mostly focuses on the performance of LLMs in general code generation tasks, with limited evaluation specifically for the field of Web development. This gap in research makes it difficult to optimize and improve the performance of LLMs in Web generation tasks. To address this issue, this paper conducts a multi-dimensional investigation and analysis of LLM performance in Web site generation tasks. Specifically, Web generation requirements are first collected, and effective prompt engineering is designed to ensure the accurate fulfillment of user expectations. These prompts are then input into multiple LLMs to initiate a self-iteration process, gradually optimizing the generated code to achieve the expected functionality as much as possible. Next, the generated code is fed back into the LLMs for security self-iteration. At this stage, the models perform vulnerability detection and repair on the code they have generated to enhance its security. After this process, the generated code enters the manual review phase, where predefined quantitative metrics are applied to evaluate and obtain corresponding indicator values. Finally, through testing, the performance of different LLMs in generating both front-end and back-end Web code is analyzed, focusing on the models' generation capabilities, self-repair abilities, code security, and potential defects.

The research in this paper shows that LLM systems exhibit significant advantages in Web generation tasks. For example, in generating high-quality front-end code, LLMs are able to produce HTML, CSS, and JavaScript code that meets both semantic and functional requirements according to user demands. In terms of back-end code generation, LLMs are also able to generate well-structured, fully functional server-side logic code. However, despite the high potential demonstrated by LLMs in Web generation tasks, there is still room for improvement in certain aspects. For instance, in terms of code security, LLMs may overlook certain potential security vulnerabilities; when handling complex business logic, LLMs may produce incomplete functions or encounter logical errors.

2. Background

2.1. LLM-based Code Generation

Current research on LLM-based code generation has made significant progress in several areas. In terms of code generation quality, research has notably improved the accuracy and coverage of code generation through the simulation of software process models (such as FlowGen) [4] and multi-agent collaboration (such as ClarifyGPT) [5]. Regarding syntactic correctness, frameworks like SynCode [6] have effectively reduced syntax errors in the generated code through syntactic decoding. In terms of ethics and fairness, studies have highlighted potential biases in code generation and proposed

DOI: 10.54254/2753-8818/100/2025.21617

prompt-based bias mitigation strategies (such as one-shot and few-shot learning) [7]. At the same time, solutions to problems like vague requirements and non-determinism have been proposed, such as clarification question generation and temperature parameter optimization, further improving the reliability of generated code [5]. For evaluating the quality of LLM-generated code, the EvalPlus framework enhances existing benchmarks by automatically generating a large number of test cases, combining LLM-based and mutation-based strategies to generate test inputs, significantly improving the rigor of the evaluation [8]. Moreover, the outstanding code generation and understanding capabilities of LLMs have been widely applied in multiple fields: for example, LLM-based code generation methods (such as generating high-quality test cases through filtering strategies and seed scheduling) have been used to test compilers, significantly improving test coverage and code quality [9] ; [10] explored an LLM plugin integrated into IDEs designed to assist developers in better understanding code through automated queries, enhancing task completion efficiency. However, these studies have not fully considered the interpretability and dynamic adaptability of code generation. There remain limitations in terms of semantic correctness and real-time interactive requirements in complex scenarios, and further exploration is needed to improve the interpretability, dynamic adaptability, and robustness of code generation in practical applications.

2.2. Intelligent Web Development

Automated web page generation and automated web page testing have always been popular and important research topics. [11] proposed a Web application development method based on Model-Driven Architecture (MDA) and UML, which uses the automated code generation tool xGenerator to convert UML models into Java code, simplifying the development process and improving development efficiency. [12] proposed a new method for automatically generating web page objects through reverse engineering, reducing manual coding work and improving the readability and maintainability of end-to-end test code.

Large language models (LLMs) are profoundly changing the paradigm of web code development, providing developers with entirely new tools and methods. In terms of web page generation, LLMs can automatically generate web-related code, significantly reducing development complexity [13]. However, the security of the generated code still requires significant attention. Research shows that PHP code generated by GPT-4 contains numerous vulnerabilities, such as insecure file uploads, SQL injections, and XSS attacks, some of which can be directly exploited, posing significant threats to software security [14]. In the task of understanding HTML code, research indicates that fine-tuned LLMs improved accuracy in semantic classification tasks by 12%, demonstrating their potential in parsing and manipulating HTML code [15]. Additionally, in web accessibility evaluation, LLMs have shown superior capabilities over traditional automation tools, effectively detecting accessibility issues requiring human intervention (such as non-text content, link purposes, and some language issues), with an overall detection rate of 87.18%, filling the gaps of existing automation tools [16].

However, there are still important gaps in current research. First, there is a lack of systematic research on the design of prompts for large language models in generating web pages. Existing work has yet to explore in-depth how to optimize prompts to more accurately convey requirements, thereby improving code generation quality. Second, in terms of code security, the self-iteration capabilities of large language models have not been sufficiently validated, especially when directed toward specific prompts; their ability to fix web code security vulnerabilities still lacks experimental support. Furthermore, there is currently no systematic evaluation method for the entire web code generation process (including prompt design, code generation, security vulnerability detection, and repair), and no horizontal comparison of the performance of different large language models in web code generation capabilities has been conducted. Addressing these issues will provide an important foundation for further enhancing the application value and reliability of LLMs in web development.

DOI: 10.54254/2753-8818/100/2025.21617

3. Approach

3.1. Workflow

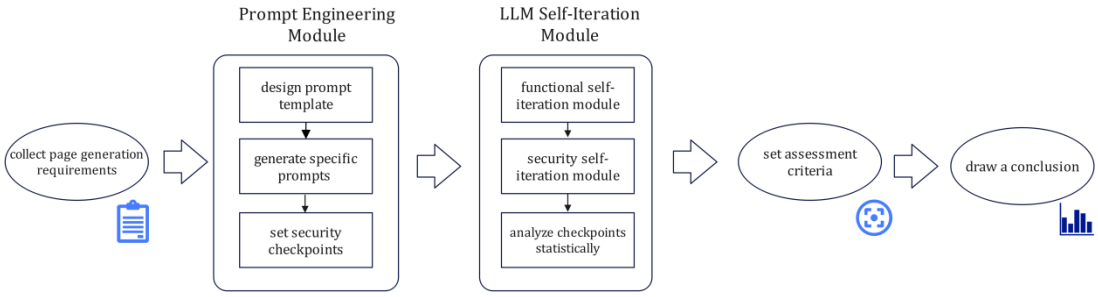

Figure 1: Workflow of Web Generation Evaluation for LLMs

The workflow of this paper is shown in Figure 1. First, diverse page generation requirements are collected, and specific prompts are generated for each requirement through prompt engineering. These prompts are then input into different LLMs to initiate the self-iteration process. The iteration stops when the model output converges or the generated code meets the requirements. After completing the iteration, the model outputs HTML, CSS, and JavaScript code. Next, these codes are re-entered into the LLM for security self-iteration, meaning the model detects and fixes vulnerabilities in the code it generated. The security-enhanced code then enters the manual review stage, where it is evaluated using preset quantitative indicators, generating numerical values for various metrics. Finally, these values undergo statistical analysis, and the performance of different models is compared.

3.2. Prompt Engineering Module

Based on the collected page generation requirements, related prompt designs are created. The prompt template is as follows:

> Please design a <page type> containing <number of sections> sections, with the first section being <section name> located <position> and using a <layout format> with <font size> and <color tone>. The following functionalities and user interactions are required: <function requirements><user interactions>. <Subsequent section descriptions>
>
> The security conditions to be implemented are as follows: <security requirements>.

Where the modules are defined as follows:

(1) Page type: homepage, about us, product or service page, contact page, FAQ page, blog or news page, case studies or customer reviews page, privacy policy page, terms of service page, search results page, 404 error page, user dashboard page, events or promotions page, download page, registration page, user feedback or comment page, forum or community page, multimedia page, map or location page, subscription page, member area, temporary page, comparison page, donation or support page, etc.

(2) Number of sections: n

(3) Position: below <section name>, to the left of <section name>, at the bottom of the page, etc.

(4) Layout format: full-width banner, grid layout, horizontal layout, list layout, icon and text side-by-side layout, form layout, paragraph layout, etc.

(5) Font size: optional, in px.

(6) Color tone: optional, colors.

DOI: 10.54254/2753-8818/100/2025.21617

(7) Function requirements: optional, based on page design needs.

(8) User interactions: optional, based on page design needs.

(9) Security requirements: optional, include common security issues (e.g., XSS, CSRF, SQL injection).

For example:

"Please design a homepage for an e-commerce website, containing 5 sections, with the first section being a carousel located at the top of the page, using a full-width banner layout, no specific font size, and a bright color tone (e.g., blue, orange). The required functionalities and user interaction scenarios are as follows: automatic carousel and manual switching functions, display of the latest products and promotions, each carousel image clickable to link to relevant product or promotion pages, users can manually switch images using the left and right arrows, and a brief description and 'Buy Now' button are shown when hovering the mouse over the image. The second section is a popular products area, located below the carousel, using a grid layout (3 columns), with a title font size of 18px, description font size of 14px, and a warm color tone (e.g., yellow, orange). It should display 3-6 popular products, including product images, names, prices, and 'Add to Cart' buttons. When hovering over the products, detailed information is shown. Users can click on the product images for more details, and the 'Add to Cart' button adds the product to the cart and displays a confirmation prompt. <Other section descriptions>

The security requirements are as follows: Ensure all user inputs (such as search boxes and contact forms) are validated to prevent XSS attacks, all data transmission uses the HTTPS protocol, and user information is secure."

This set of prompts aims to achieve the most accurate description of page layout and functionality in natural language. The prompt's specificity covers each section's function, position, layout, and related features, reasonably limiting the LLM's freedom and effectively conveying the user's web requirements. By striving to convey all requirements in the initial description, this approach reduces the number of subsequent prompts and simplifies the design of follow-up prompts. Additionally, this method provides a reasonable baseline for the later evaluation of the LLM's code generation capability and lays the foundation for the objective design of quantitative evaluation indicators.

Furthermore, during the LLM's subsequent iteration process, prompt design will also be involved. For models with contextual abilities, only certain prompts need to be given, for example: "Please assess the security issues in the above code and provide the fixed code."

When the iteration converges, certain prompts can be given in some areas, such as the design of prompts related to security vulnerabilities as follows:

> The above code has vulnerabilities in <security vulnerability>, <specific prompt>.

<Security vulnerability> includes input validation and filtering, cross-site scripting (XSS) protection, cross-site request forgery (CSRF) protection, data storage and transmission security, security of third-party dependencies, access control and authentication, error handling and logging, browser security policies, code quality and maintainability, user privacy protection, and other areas.

<Specific prompt>: Optional, specify the particular area based on the situation.

3.3. LLM Self-Iteration Module

The LLM self-iteration module primarily consists of the functional self-iteration module and the security self-iteration module. The main task of the functional self-iteration module is to generate code that meets functional requirements based on the prompt. First, the prompt is input into the LLM to generate initial code. The generated code is then reviewed to check whether it fully satisfies all the requirements in the prompt, including functional implementation and detailed specifications. If the

DOI: 10.54254/2753-8818/100/2025.21617

code meets all the requirements, iteration stops; if there are unmet requirements, the iteration continues, and during the iteration process, errors or unimplemented features are clearly pointed out to guide the model in making corrections. There are two termination conditions for the iteration: first, when the code fully meets all the details and functional requirements of the prompt; second, if after multiple iterations the LLM still cannot implement a certain feature, iteration stops and the current best code is output. After the iteration ends, the final web code is output, without checking for security or vulnerabilities at this stage.

The task of the security self-iteration module is to conduct a comprehensive security and quality assessment of the code generated by the LLM. First, the final code generated by the LLM is input into the security self-iteration module for an initial quality check and security issue assessment. In each iteration, no specific directional prompts are provided; instead, a comprehensive quality check and security issue assessment are required, and necessary modifications are made based on the evaluation results. When the iteration converges (i.e., when the code quality and security reach a stable state), it is checked whether the pre-set security checkpoints for the code have been modified. For any incomplete security points, directional prompts are given, and iteration continues until all security checkpoints are satisfied. Finally, the code that is in the best state in terms of security, readability, and performance is output.

After both modules have concluded, the implementation status of the relevant checkpoints is statistically analyzed. Section 4 will provide a detailed explanation of the quantitative indicator evaluation and results comparison module.

For example, using DeepSeek, after generating the webpage code based on the prompt, a check reveals that the user comment refresh functionality was not implemented. At this point, the prompt system displays: “The current code does not meet all functional requirements, please make corrections.” After adjusting the code, it is confirmed that the functionality has been implemented, and a check ensures no missing features. Next, the code enters the security self-iteration module, where the system continuously prompts: “Please assess the security issues in the above code and provide the fixed code.” This process will continue until the security evaluation converges. Finally, the number of fixed security issues is assessed, and prompts are given for any unresolved security points to evaluate the model’s ability to address them.

4. Evaluation

4.1. Requirement

We have set up a web generation quality evaluation task, which is comprehensively assessed from the following four dimensions:

1) Functional Accuracy

This dimension aims to evaluate whether the generated web page accurately reflects the functional and user interaction requirements specified in the prompt, in order to assess the model’s understanding and execution ability of the instructions. In the evaluation, 10 pages are generated, covering 139 functional points and user interaction requirements. This broad coverage enables a comprehensive assessment of the model’s performance in web generation tasks.

2) Aesthetic Quality

Due to the subjectivity of aesthetic quality, this evaluation focuses solely on whether the model can correctly respond to the design requirements explicitly specified in the prompt, including column positions, layout formats, font sizes, and color tone requirements, rather than evaluating or scoring the subjective beauty of the page. If each color tone, font size, layout, and position for every column in each page requirement is considered a standard point, this dimension covers 63 standard points in total.

DOI: 10.54254/2753-8818/100/2025.21617

3) Error Identification and Correction Ability

This dimension evaluates whether the model can autonomously identify functional errors and vulnerabilities in the generated code and possesses the ability to correct these errors. The model's performance in error identification and correction during the iteration process is observed to assess its capabilities in code generation and optimization.

4) Security Vulnerability Identification and Repair Ability

This dimension assesses whether the model can identify and repair security vulnerabilities in the generated code, thereby reducing the risk of potential attacks on the web page and ensuring user data security. In the evaluation, the initial code of 10 pages (i.e., code that has not undergone security optimization) is analyzed, revealing 43 vulnerabilities, including common risks such as XSS, CSRF, SQL injection, and data leakage. Through manual evaluation, it is determined whether the model can successfully repair these vulnerabilities after the security iteration has converged and whether it can accurately assess vulnerability risks based on the relevant prompts.

4.2. Metrics

Based on the web evaluation task and requirements, the following quantitative standards are provided for the four evaluation dimensions:

1) Functional Accuracy (Fun)

$$Fun = \sum_{i=1}^{n} F_i * \varepsilon_{fi} / T_f$$

In this formula, F_1, F_2,..., F_n represent the number of functional points achieved by the LLM during the first, second, and n-th code generation iterations, respectively. T_f represents the total number of functional points, and ε_{fi} is the decay coefficient.

2) Aesthetic Quality (Aes)

$$Aes = \sum_{i=1}^{n} A_i * \varepsilon_{ai} / T_a$$

Here, A_1, A_2,..., A_n represent the number of standard points achieved by the LLM during the first, second, and nth code generation iterations, respectively. T_a represents the total number of standard points, and ε_{ai} is the decay coefficient.

3) Error Identification and Correction Ability (Err)

$$Err = \frac{\sum_{i=1}^{n} (A_i + F_i) * \varepsilon_{ei}}{T_f + T_a - A_1 - F_1}$$

In this formula, ε_{ei} is the decay coefficient.

4) Security Vulnerability Identification and Repair Ability (Sec)

$$Sec = \frac{S_v + kS_a}{T_s}$$

Here, S_v represents the vulnerability points resolved by the LLM after self-iteration without explicit prompts, and S_a represents the vulnerability points that the LLM can resolve after receiving vulnerability prompts for security vulnerabilities that were not identified in the self-iteration. T_s represents the total number of vulnerability points, and k is the weight coefficient.

DOI: 10.54254/2753-8818/100/2025.21617

4.3. Experimental Setup

The LLMs selected for this experiment include Wenxin LLM 4.0 Turbo, Kimi, Doubao, and DeepSeek-V3. Wenxin LLM 4.0 Turbo is an efficient inference and multimodal model launched by Baidu, particularly suitable for scenarios with high real-time requirements. Kimi is a long-text processing model developed by the Dark Side of the Moon, specifically designed for dialogue systems and document summarization tasks. Doubao is a multilingual and vertical domain-optimized model developed by ByteDance, suitable for multilingual and domain-specific tasks. DeepSeek is a code generation and optimization model launched by DeepSeek, focused on developer-related tasks.

In this experiment, the values of T_f, T_a and T_s are set to 139, 63, and 43, respectively. The decay coefficient is set to $\varepsilon_{fi} = \varepsilon_{ai} = 1/n$ $\varepsilon_{ei} = 1/(n-1)$. Since the focus of this experiment is to evaluate the LLM's complete ability to resolve security vulnerabilities, it is considered that the vulnerabilities have been fully resolved as long as the code reaches the security requirements after as many iterations as possible, regardless of whether the fix was completed through model self-iteration or with the help of additional prompts. Therefore, the value of k is set to 1 to measure the final security level of the code.

4.4. Results

Table 1: LLM Comparison Evaluation Results

Metric	Wenxin LLM 4.0 Turbo	Kimi	Doubao	DeepSeek-V3
Fun	0.46	0.82	0.62	**0.89**
Aes	0.73	0.97	0.86	**0.99**
Err	0.28	0.79	0.43	**0.91**
Sec	0.41	0.67	0.48	**0.83**

As shown in Table 1, the DeepSeek-V3 model performs best in terms of functionality, aesthetics, error correction, and security vulnerability repair, while Wenxin LLM 4.0 Turbo performs the worst.

In terms of functional accuracy, DeepSeek-V3 and Kimi can cover most of the functional points during the first code generation, and the remaining functional points are usually correctly implemented in the second generation. In contrast, Wenxin LLM 4.0 Turbo can only implement a few functional points during the first generation and requires multiple prompts and iterations (an average of 3 times) to reach convergence, showing insufficient performance in implementing complex functions. In terms of page layout aesthetics, Kimi, Doubao, and DeepSeek-V3 can all implement the page layout, font, and tone requirements specified in the prompt well. Wenxin LLM 4.0 Turbo can also meet these needs after several iterations. In terms of error identification and correction ability, DeepSeek-V3 stands out, as it can autonomously identify the parts of the code that do not meet the requirements and make corrections. Kimi occasionally misses some function implementations or does not fully correct errors during the self-iteration process. Wenxin LLM 4.0 Turbo and Doubao have limited error correction abilities. In terms of security vulnerability identification and repair ability, DeepSeek-V3 can identify most of the potential risks after several rounds of iteration and propose basic repair solutions, although some vulnerabilities are not completely fixed. Wenxin LLM 4.0 Turbo, Kimi, and Doubao perform relatively poorly in security vulnerability identification, often missing some security risks.

Overall, LLMs show certain abilities in page functionality implementation, layout design, and detail handling. Among them, DeepSeek-V3 performs particularly well in error identification and

DOI: 10.54254/2753-8818/100/2025.21617

correction, significantly improving the efficiency of web generation tasks. However, LLMs still have certain limitations in security vulnerability identification and code security enhancement.

5. Conclusion

This study focuses on the comprehensive evaluation of LLMs' web generation capabilities, revealing their powerful code generation abilities. Through the design of effective prompt engineering, initiating self-iteration processes, and performing security self-iterations, along with the quantification of metrics and standard reviews, the conclusion is drawn that LLMs can effectively implement the page functionality and layout requirements described in the prompt for web generation tasks, demonstrating high generation capability. However, in terms of web code security, LLMs still have room for improvement. Future research directions could focus on enhancing code security, specifically by introducing more comprehensive security vulnerability detection mechanisms, combining static analysis tools (such as SonarQube) and dynamic testing tools (such as OWASP ZAP); training LLMs to identify more types of security vulnerabilities (such as XSS, CSRF, SQL injection, etc.), and generating more thorough repair solutions.

References

[1] *Jiang, J., Wang, F., Shen, J., et al. (2024). A survey on large language models for code generation. arXiv preprint arXiv:2406.00515.*

[2] *Sharma, M., & Angmo, R. (2014). Web-based automation testing and tools. International Journal of Computer Science and Information Technologies, 5(1), 908-912.*

[3] *Kaluarachchi, T., & Wickramasinghe, M. (2023). A systematic literature review on automatic website generation. Journal of Computer Languages, 75, 101202.*

[4] *Lin, F., & Kim, D. J. (2024). When LLM-based code generation meets the software development process. arXiv preprint arXiv:2403.15852.*

[5] *Mu, F., et al. (2023). ClarifyGPT: Empowering LLM-based code generation with intention clarification. arXiv preprint arXiv:2310.10996.*

[6] *Ugare, S., et al. (2024). Improving LLM code generation with grammar augmentation. arXiv preprint arXiv:2403.01632.*

[7] *Huang, D., et al. (2023). Bias assessment and mitigation in LLM-based code generation. arXiv preprint arXiv:2309.14345.*

[8] *Liu, J., Xia, C. S., Wang, Y., et al. (2024). Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. Advances in Neural Information Processing Systems, 36.*

[9] *Gu, Q. (2023). LLM-based code generation method for Golang compiler testing. In Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2023) (pp. 2201-2203). Association for Computing Machinery. https://doi.org/10.1145/3611643.3617850*

[10] *Nam, D., Macvean, A., Hellendoorn, V., et al. (2024). Using an LLM to help with code understanding. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (pp. 1-13).*

[11] *Paolone, G., Marinelli, M., Paesani, R., et al. (2020). Automatic code generation of MVC web applications. Computers, 9(3), 56.*

[12] *Stocco, A., Leotta, M., Ricca, F., et al. (2017). APOGEN: Automatic page object generator for web testing. Software Quality Journal, 25(3), 1007-1039.*

[13] *Calò, T., & De Russis, L. (2023). Leveraging large language models for end-user website generation. In International Symposium on End User Development (pp. 52-61). Springer Nature Switzerland.*

[14] *Tóth R., Bisztray T., Erdődi L. (2024). LLMs in web development: Evaluating LLM-generated PHP code unveiling vulnerabilities and limitations. In International Conference on Computer Safety, Reliability, and Security (pp. 425-437). Springer Nature Switzerland.*

[15] *Gur, I., Nachum, O., Miao, Y., et al. (2022). Understanding HTML with large language models. arXiv preprint arXiv:2210.03945.*

[16] *López-Gil, J. M., & Pereira, J. (2024). Turning manual web accessibility success criteria into automatic: An LLM-based approach. Universal Access in the Information Society, 1-16.*