electronics

MDPI

Article

An Automatic Code Generation Tool Using Generative Artificial Intelligence for Element Fill-in-the-Blank Problems in a Java Programming Learning Assistant System

Zihao Zhu [1,*], Nobuo Funabiki [1,*], Mustika Mentari [1], Soe Thandar Aung [1], Wen-Chung Kao [2] and Yi-Fang Lee [3]

[1] Department of Information and Communication Systems, Okayama University, Okayama 700-8530, Japan; pqt85hm5@s.okayama-u.ac.jp (M.M.); soethandar@s.okayama-u.ac.jp (S.T.A.)
[2] Department of Electrical Engineering, National Taiwan Normal University, Taipei 10610, Taiwan; jungkao@ntnu.edu.tw
[3] Department of Industrial Education, National Taiwan Normal University, Taipei 10610, Taiwan; ivana@ntnu.edu.tw
* Correspondence: pjco5fyn@s.okayama-u.ac.jp (Z.Z.); funabiki@okayama-u.ac.jp (N.F.)

Abstract: Presently, *Java* is a fundamental object-oriented programming language that can be mastered by any student in information technology or computer science. To assist both teachers and students, we developed the *Java Programming Learning Assistant System (JPLAS)*. It offers several types of practice problems with different levels and learning goals for step-by-step self-study, where any answer is automatically marked in the system. One challenge for teachers that is addressed with *JPLAS* is the generation of proper exercise problems that meet learning requirements. We implemented programs for generating new problems from given source codes, as collecting and evaluating suitable codes remains time-consuming. In this paper, we present an *automatic code generation tool* using *generative AI* to solve this challenge. Prompt engineering is used to help generate an appropriate source code, and the quality is controlled by optimizing the prompt based on the outputs. For applications in *JPLAS*, we implement a web application system to automatically generate an *element fill-in-the-blank problem (EFP)* in *JPLAS*. For evaluation, we select the *element fill-in-the-blank problem (EFP)* as the target type in *JPLAS* and generate several instances using this tool. The results confirm the validity and effectiveness of the proposed method.

Keywords: JPLAS; Java programming learning; learning requirements; generative AI; prompt engineering; quality control; prompt optimization

Academic Editor: Manuel Mazzara

Received: 22 March 2025
Revised: 8 May 2025
Accepted: 12 May 2025
Published: 31 May 2025

Citation: Zhu, Z.; Funabiki, N.; Mentari, M.; Aung, S.T.; Kao, W.-C.; Lee, Y.-F. An Automatic Code Generation Tool Using Generative Artificial Intelligence for Element Fill-in-the-Blank Problems in a Java Programming Learning Assistant System. *Electronics* **2025**, *14*, 2261. https://doi.org/10.3390/electronics14112261

1. Introduction

Java has remained one of the most popular programming languages since its inception in 1995. From embedded systems to web development, big data, and cloud computing, *Java* is a vast and mature ecosystem. Its extensive libraries and robust frameworks have made it the preferred choice for enterprise development. *Java* developers continue to be in high demand worldwide [1], making the systematic learning of *Java* as essential as ever.

In practice, the learning of *Java* presents challenges for both students and teachers. Beginners often find abstract concepts in *object-oriented programming* difficult to understand. Teachers struggle to design effective exercise problems that help students grasp key concepts.

To address this issue, we developed the web-based *Java Programming Learning Assistant System (JPLAS)* [2]. *JPLAS* offers several types of Java programming exercise problems

ranging from studying code reading and syntax practice to hands-on coding using *object-oriented programming* concepts. With *JPLAS*, teachers can upload prepared exercise problems, and students can study and review them in the system at their own pace.

Currently, *JPLAS* provides *grammar-concept understanding problems* [3], *element fill-in-the-blank problems* [4], *code completion problems*, *value trace problems*, *statement element fill-in-the-blank problems*, and *code writing problems*.

To reduce the load for teachers in preparing problems, we developed programs that automatically generate new problems from given appropriate source codes [4]. However, teachers still need to collect and evaluate source codes that align with their teaching contents, which remain a time-consuming tasks. Fortunately, with the development of generative AI models such as *chatGPT* [5], source codes can be generated automatically; however, due to several issues including inaccurate prompts or unclear evaluation methods, the source codes generated by AI models sometimes do not meet the requirements of novices for learning programming.

In this study, we propose an *automatic code generation tool* based on *generative AI*, which leverages *prompt engineering* techniques to assist in generating high-quality source code suitable for novice learners. Unlike approaches that require model fine-tuning, our method enhances code quality by analyzing the AI model's output and optimizing the input prompts accordingly. We systematically compare three prompting strategies—*zero-shot*, *chain-of-thought (CoT)*, and *black-box prompt optimization (BPO)* [6]—across three representative models: *ChatGPT-4o* [7], *DeepSeek-R1-7B* [8], and *LlaMA3.2-1B* [9].

The experimental results indicate that, in resource-constrained environments, *mid-range models* such as *DeepSeek-R1-7B* combined with CoT and BPO strategies provide a strong balance between performance and efficiency. In contrast, *high-performance models* like *ChatGPT-4o* deliver the best results when the application demands robustness across a wide range of task difficulties. *Lightweight models* such as *LlaMA3.2-1B*, while limited in capability, still prove useful in scenarios with very limited computational resources or for basic-level tasks. Moreover, user study experiments with student participants validate the system's educational effectiveness.

As a practical application of this tool in the *JPLAS* educational platform, we implement a web-based system for automatically generating *element fill-in-the-blank problems (EFPs)*. The system integrates various open-source components including *Spring Boot* [10], *jQuery* [11], *Ollama* [12], and *Docker* [13]. Functional testing on ten beginner-level topics confirms that, with well-crafted prompts, even small AI models can efficiently generate high-quality EFP instances on a PC.

We emphasize that our system offers the following key advantages: it supports *fully offline deployment* (ensuring data privacy), is suitable for *low-resource educational environments*, and improves code generation quality through prompt engineering without requiring expensive model fine-tuning. Overall, this work provides a *flexible, secure, and locally deployable AI-based solution* for programming education, particularly suited for localized and personalized learning scenarios.

The rest of this paper is organized as follows: Section 2 introduces the related works in the literature. Section 3 reviews our preliminary work leading to this paper. Section 4 proposes the methodology of our system. Section 5 shows the evaluation results of this method. Section 6 presents the implementation of a web application system for EFP instance generation. Finally, Section 7 concludes this paper with a discussion of future work.

2. Related Works

In this section, we introduce some related works in the literature.

2.1. Programming Education

First, we introduce works on programming education.

In [14], McGill et al. proposed a framework for analyzing students' knowledge of programming and pointed out that, in programming education assessments, students' mastery should be evaluated from three dimensions: syntactic, conceptual, and strategic.

In [15], Altadmri et al. collected error messages and Java code from around 265,000 users worldwide through a black-box project. By analyzing these novice students' mistakes, their research effectively helped to understand students' misconceptions and guide the design of programming teaching methods.

In [16], Gomes et al. highlighted that the challenges students faced in learning computer programming were primarily due to a lack of problem-solving and algorithm design skills. They reviewed several educational tools that have successfully been used to support programming education and proposed a new computational system aimed at further addressing common learning difficulties.

In [17], Sorva et al. conducted a comprehensive survey of the generic program visualization systems designed to teach novice programmers about program runtime behavior. They covered the tools developed over three decades, examined their features and empirical evaluations, and discussed the role of learner engagement. The authors found that many systems were short-lived prototypes, although recent trends aimed to support more interactive and engaging user experiences. While the evaluation results generally supported the use of program visualization in introductory programming education, the research at the time was insufficient to draw detailed conclusions about learner engagement. Based on their review, the authors proposed future research directions related to engagement, cognitive load, task authenticity, and integration into programming pedagogy.

In [18], Medeiros et al. conducted a systematic review on teaching and learning introductory programming in higher education, focusing on more recent research from 2010 to 2016. They proposed a categorization of the key challenges faced by students and teachers, emphasizing that problem-solving and mathematical ability are critical for students' success. The review identified persistent difficulties such as low motivation, challenges with programming syntax, and the lack of scalable and personalized teaching methods, highlighting areas for future research and improvement.

In [19], Lindberg et al. investigated programming education guidelines in K–12 curricula across seven countries, reviewed commercially available programming-related games, and compared the findings to suggest appropriate age groups for these games. Their results provided insights for educators and curriculum designers interested in gamifying programming education.

In [20], Olsson et al. explored how visualization and gamification enhanced learner control and motivation in online programming courses. Their research, conducted through evaluations and group discussions in two Moodle-based courses, showed that progress bars helped students navigate complex course content, although the effectiveness of such visualization varied with learner style. They also found that the digital badges used for gamification had different motivational effects across study groups, with traditional grades remaining more influential in conventional programs. The study concluded that software visualization could be a promising approach to improve programming education.

Based on the above literature review, several prominent trends in the research on and practice of programming education can be identified. Firstly, an increasing number of studies are emphasizing students' cognitive misconceptions and learning difficulties in programming education, particularly in problem-solving skills, algorithm design abilities, and mastery of programming syntax. By analyzing students' errors and feedback, researchers are able to provide valuable data to improve programming teaching methods. Additionally, program visualization and gamification are widely regarded as effective ways

to enhance student engagement and motivation, especially among beginners, helping them better understand program execution and master programming concepts.

In the future, programming education will place greater emphasis on diversified learning tools, personalized teaching methods, and active student participation. The goal is to cultivate programming talent with advanced thinking skills and the ability to solve real-world problems. As programming education evolves, it will increasingly focus on fostering student engagement through interactive and personalized learning experiences to develop students' higher-order thinking and problem-solving capabilities.

2.2. Generative AI

Second, we introduce the works on generative AI used in programming educations.

In [21], Luckin et al. found that by establishing a suitable AI model, educational content can be made more engaging and effective to realize the goal of personalized education.

In [22], Chen et al. assessed the impact of AI on education, ascertaining that AI has been extensively adopted in education, has helped teachers work more effectively, and has enhances overall teaching quality.

In [23], Baidoo-Anu1 et al. studied some potential benefits and drawbacks of *ChatGPT* in promoting teaching and learning and offered recommendations on how *ChatGPT* could be leveraged to maximize its impact on teaching and learning.

Coding Rooms [24] is an online platform designed for real-time programming education. It allows instructors to create interactive coding assignments, conduct live coding sessions, and monitor student progress in real time. The platform supports various programming languages such as *Python*, *Java*, and *C++*. In recent updates, *Coding Rooms* has integrated AI features to enhance both teaching and learning experiences. These AI tools can automatically evaluate student submissions, provide instant feedback, detect common mistakes, and even offer code hints or explanations. Additionally, instructors can use AI to generate test cases or grading rubrics, improving scalability and efficiency in classroom settings.

Khan Academy [25] is a well-known nonprofit educational platform offering free courses in subjects such as mathematics, science, and programming. In recent years, it has integrated AI into its platform through a tool called *Khanmigo* [26], developed in collaboration with OpenAI. *Khanmigo* functions as an AI-powered virtual tutor and classroom assistant. It uses *large language models (LLMs)* to provide personalized feedback, help students understand complex concepts, and support teachers in managing classroom interactions. The AI tutor can simulate Socratic questioning, assist with debugging code, and offer step-by-step guidance tailored to a learner's level.

GitHub Copilot [27] is an AI-powered coding assistant developed by GitHub in collaboration with OpenAI. It leverages the Codex language model to provide real-time code suggestions, auto-completions, and function generation within widely used code editors such as *Visual Studio Code* and *JetBrains*. *Copilot* supports multiple programming languages, including *Python*, *JavaScript*, *Java*, and *C++*. In terms of AI application, *GitHub Copilot* demonstrates the integration of LLMs into programming environments. It utilizes natural language understanding to interpret code comments and generate relevant code snippets. Additionally, it learns from the surrounding context and user behavior to offer intelligent and context-aware suggestions. This enhances coding efficiency and supports learners in solving programming tasks with minimal prior knowledge.

Based on the above literature review, AI-assisted systems each have distinct characteristics and advantages. Firstly, *ChatGPT* has demonstrated a high accuracy rate in answering user questions, providing students with accurate and timely feedback, particularly when addressing programming-related queries. Its natural language processing capabilities are especially impressive. *GitHub Copilot*, on the other hand, is more focused on programming

assistance, enhancing coding efficiency through auto-completion and code suggestions. It effectively reduces errors and helps students understand and master programming languages, particularly for common programming tasks. Additionally, platforms like *Coding Rooms* and *Khanmigo*, by integrating AI features, can generate personalized practice problems in real time based on students' learning progress and needs, which offers a significant advantage in personalized education. However, the common issue across these systems is the high consumption of computational resources, and their responsiveness and accuracy may be affected in the absence of a stable network.

After analyzing the characteristics of existing AI-assisted systems, we also need to consider an important issue—data security and privacy protection. Many current AI systems rely on cloud computing to provide real-time feedback and suggestions. This means that all data (such as students' code, problem descriptions, learning progress, etc.) are transmitted to servers for processing and storage, which introduces certain risks of data breaches and privacy leaks. For educational institutions, especially when dealing with students' personal information and learning data, this cloud-based processing approach could pose significant security risks. Therefore, offline AI has become an important solution. By localizing AI models, educational systems can avoid uploading sensitive data to external servers, thus reducing the risk of data leakage.

Our proposed system focuses on school-level, self-developed learning tools and is designed to address core limitations such as the dependency on online services, high computational requirements, the lack of customization, and privacy concerns.

2.3. Prompt Engineering

Third, we introduce the works on prompt engineering.

In [6], Chen et al. highlighted the crucial role of prompt engineering in enhancing the capabilities of *large language models (LLMs)* as well as explored both fundamental and advanced prompt engineering techniques, such as self-consistency, chain-of-thought (CoT), and generated knowledge.

In [28], Cheng et al. introduced *black-box prompt optimization (BPO)* as a novel alignment approach to solve the alignment problem in *LLMs* and demonstrated its effectiveness in optimizing prompts without modifying model parameters. They highlighted the advantages of *BPO* over traditional training-based methods, showing that it had a 22% and 10% higher win rate than *ChatGPT* and *GPT-4*, respectively.

In [29], Agarwal et al. proposed *PromptWizard*, a fully automated framework for discrete prompt optimization. By utilizing a feedback-driven critique and synthesis mechanism, it iteratively refines both instructions and in-context examples to improve prompt quality. *PromptWizard* demonstrates strong performance across 45 tasks, maintains effectiveness with limited data and smaller LLMs, and significantly reduces API usage and operational costs.

In [30], Fernando et al. proposed *PromptBreeder*, a self-referential self-improvement framework that evolves prompts for specific domains using LLM-driven mutation and evaluation. By simultaneously optimizing both task prompts and mutation prompts, *PromptBreeder* outperformed hand-crafted strategies like Chain-of-Thought and Plan-and-Solve on arithmetic, commonsense reasoning, and hate speech classification tasks.

Recent studies have shown that *prompt engineering* plays a crucial role in enhancing the capabilities of LLMs. Compared to traditional fine-tuning methods, *prompt engineering* offers a more flexible and cost-effective optimization approach. By designing and refining prompts, researchers can significantly improve the performance of AI models on specific tasks, while avoiding complex adjustments to model parameters.

In this study, *prompt engineering* was applied to a local AI model to generate suitable educational source code. Compared to traditional fine-tuning methods, *prompt ngineering* provides a more flexible and efficient way to optimize the AI model's performance, especially when generating code that meets specific programming requirements. By designing and optimizing specific prompts, the local AI model can be guided to generate programming code that better aligns with teaching needs, thereby helping students better understand programming concepts and improve their programming skills.

3. Preliminary Works

In this section, we review our work conducted prior to that in this study.

3.1. Java Programming Learning Assistant System (JPLAS)

We developed *JPLAS* as a software platform to support the programming teaching and learning. Students can use this platform for the independent learning of basic knowledge of *Java* programming, identifying gaps in their understanding through solving practice problems, and focusing on targeted learning. At the same time, teachers can easily collect and manage data on students' learning progress and performance.

Originally, *JPLAS* ran on a server, with practice problems stored in a database. Teachers could easily adjust the practice questions based on teaching goals and progress. At the same time, students' performances were uploaded and saved in real time, allowing both teachers and students to promptly assess learning progress.

Subsequently, to facilitate learning in environments without Internet access, we developed an offline version of *JPLAS*. This version was deployed using *Docker* containers, with *LocalStorage* of *JavaScript* replacing the database. Additionally, the answers to the problems were encrypted using the *SHA256 algorithm*.

JPLAS provided various types of problems for students, ranging from *grammar-concept understanding problems (GUPs)* and *value trace problems (VTPs)*, focusing on reinforcing fundamental concepts, to *element fill-in-the-blank problems (EFPs)*, *code completion problems (CCPs)*, and *code writing problems (CWPs)*, helping develop correct programming habits. This step-by-step learning approach in *JPLAS* has proven effective in helping beginners to self-learn *Java* programming concepts, in cultivating programming thinking, and in assisting teachers with targeted instruction.

3.2. Element Fill-in-the-Blank Problem

In this paper, we select the *EFP* as the application target of the proposed code generation tool for *Java* programming learning. An *EFP* instance requests students to fill in the blank elements in the given source code. Among all the exercise problems, the creation of *EFPs* is relatively challenging. It is necessary to choose a source code that aligns with the target knowledge points and has an appropriate difficulty level to achieve the best teaching results. At the same time, the source code must meet certain requirements that are necessary for question generation. Therefore, collecting a sufficient number of suitable source codes has become a major challenge.

Here, we give a detailed introduction to the *EFP* as well as generation algorithms from a source code.

3.2.1. Definition of an Element

An element is the smallest unit of a source code, consisting of a *reserved word*, an *identifier*, a *control symbol*, or an *operator*.

- **Reserved words** are predefined sequences of characters that serve specific functions, such as "private" or "public".

- **Identifiers** are names defined by the programmer to represent variables, classes, or methods.
- **Control symbols** include punctuation marks used in the syntax, such as "." (dot), ":" (colon), ";" (semicolon), "()" (parentheses), and "{}" (curly brackets).
- **Operators** are used in conditional expressions to define logical conditions, such as "<" and "&&".

3.2.2. Blank Element Selection Algorithm

The *blank element selection algorithm* identifies the maximum number of feasible blank elements in a given source code using graph theory [4].

In the first step, the algorithm constructs a *compatibility graph* by treating each candidate blank element as a *vertex*. An *edge* is created between two vertices if they can be blanked simultaneously. To ensure correctness, certain conditions are defined to prevent elements from being blanked together to avoid ambiguity.

In the second step, the algorithm extracts a *maximal clique* from the compatibility graph to determine the largest possible set of blank elements. It is obvious that the *EFP* difficulty increases as more elements are blanked. By selectively blanking a subset of these elements, a variety of fill-in-the-blank problems can be generated at different difficulty levels. The algorithm follows these detailed steps:

1. **Vertex generation for the constraint graph:** each potential blank element is selected from the source code and represented as a vertex in the constraint graph.
2. **Edge generation for the constraint graph:** an edge is added between any two vertices that should not be blanked simultaneously to ensure uniqueness.
3. **Compatibility graph construction:** the complement of the constraint graph is taken to create the compatibility graph, which represents pairs of elements that can be blanked together.
4. **Clique extraction:** a maximal clique is identified using a simple greedy algorithm to select the largest possible set of blank elements with unique answers with the following steps:
    - Select the vertex with the highest degree in the compatibility graph.
    - Remove this vertex and all its non-adjacent vertices.
    - Repeat until no vertices remain.

3.2.3. Coding Rule Check Function

Coding rules define a set of conventions for writing high-quality source code. Following these rules ensures code uniformity and improves its readability, maintainability, and scalability. The coding rules cover *naming conventions*, *coding styles*, and *potential problems*. To enforce these standards, we implemented a *coding rule check function*, which automatically verifies the compliance of the source code [31].

1. **Naming rules:** Naming rules help identify naming errors in source code. We adopt the camel case as the standard *Java* naming convention:
    - **Variables, methods, and method arguments:** the first letter should be lowercase, with each subsequent word capitalized.
    - **Classes:** the first letter of each word should be uppercase.
    - **Constants:** all letters should be uppercase.
    - **Identifiers** must be meaningful English words; Japanese or Romanized Japanese should not be used.
2. **Coding styles:** Coding style rules ensure a consistent code layout by checking elements such as indentation, bracket placement, and spacing. Following these rules improves code clarity and uniformity, making it easier to read and maintain.

3. **Potential problems:** Potential issues refer to code segments that can be compiled successfully but may introduce functional errors or bugs. These include:
    - **Dead code:** portions of the code that are never executed.
    - **Overlapping code:** multiple code segments with similar structures and functions.

3.2.4. EFP Generation Steps

Using the programs described above, a valid new *EFP* instance can be generated through the following steps:

1. Select a source code that covers the syntactic topics to be studied.
2. Apply the *coding rule check function* to detect and fix the issues with naming rules, coding styles, or potential problems.
3. Apply *JFlex* and *Jay* [32] to tokenize the source code into a sequence of lexical units or elements ,and classify each element type.
4. Apply the *blank element selection algorithm* to choose the blank elements that have grammatically correct and unique answers.
5. Upload the generated *EFP* instance to the *JPLAS* server.

4. Methodology for Proper Code Generation

In this section, we present the methodology for generating proper source code for *Java* programming learning. With the increasing demand for educational content in *JPLAS*, there is a growing need to develop a method for generating a source code that is not only correct and efficient but also contains the educational content for reinforcing learning.

4.1. Adopted Approach for AI

Recent advancements in AI have significantly influenced the field of programming education. There have been many attempts to use and studies on using AI to generate content that aligns with educational materials. However, ensuring AI-generated content meets human needs remains a challenge, where existing alignment methods primarily focus on further training these models. The additional training required for *LLMs* is often expensive in terms of GPU computing resources. Additionally, some models, such as certain versions of *ChatGPT*, are not accessible for the user-demanded training. Under the situations, our study aimed to generate high-quality *Java* source code across different AI models using various prompt engineering strategies without further training.

4.2. Code Quality Assessment

First, we designed an objective function to assess the quality of AI-generated code. This function consists of four evaluation criteria:

- Code accuracy;
- Code relevance to the topic;
- Code difficulty;
- Feasibility of problem generation.

4.2.1. Code Accuracy

Code accuracy refers to the correctness of the code. First, it must comply with the standard syntax, ensuring it can be compiled and executed. Additionally, it should follow the proper coding conventions. In this paper, we use the *coding rule check function* for validation. This function automatically detect issues in *Java* code related to *naming rules*, *coding styles*, and *potential problems*.

The f_{accuracy} score is determined based on a comprehensive evaluation of the *coding rule check function* and the *JUnit* [33] test results using the following formula:

$$f_{\text{accuracy}} = \alpha \cdot S_{\text{rule}} + (1 - \alpha) \cdot S_{\text{test}}, \tag{1}$$

where

- $S_{\text{rule}} \in [0, 1]$ is the score obtained from the *coding rule check function*, calculated as the proportion of passed checks over total checks.
- $S_{\text{test}} \in [0, 1]$ is the score from the *JUnit* test results, calculated as the proportion of passed test cases.
- $\alpha \in [0, 1]$ is a tunable parameter that adjusts the relative importance of style versus functionality. Here, we set $\alpha = 0.4$ to emphasize functional correctness.

In practical applications, especially in the context of educational code, the accuracy of coding standards is crucial. Educational code serves as a learning example and must be free from errors, as it is not just executable code but also a foundation for students' learning and reference. Therefore, in addition to basic syntax rules, aspects such as code readability, naming conventions, and style consistency must be strictly enforced.

In this study, since the *JUNIT* test cases were automatically generated by *EvoSuite* [34], although it is a powerful tool, the generated test cases were not perfect: sometimes, they missed certain edge cases or failed to comprehensively cover all code paths. Therefore, a minimum pass rate threshold was set to 70%.

4.2.2. Java Learning Topics

There have been many studies on learning sequences and methods for *Java* programming language [35]. In this paper, first, we refer to the classical learning paths outlined in textbooks. Then, we divide the *Java* learning process into three stages, with each stage further broken down into multiple topics following a conventional learning sequence. Here, we give a detailed introduction to the three stages:

1. **Primary stage: Java basic grammar learning**.
    - Variables and data types;
    - Operators (arithmetic, logical, bitwise, assignment);
    - Control flow (if-else, switch-case, for-loop, while-loop, do-while-loop, break, continue);
    - Arrays (one-dimensional array, multi-dimensional array, array iteration);
    - Methods (method declaration, parameters and return values, overloading);
    - Classes and objects (class declaration, object instantiation, constructor, this, encapsulation).
2. **Intermediate stage: object-oriented and API learning**.
    - Inheritance (extends, super, override);
    - Polymorphism (upcast, downcast);
    - Abstract classes and interfaces (abstract, interface, default method);
    - Inner classes (member inner class, local inner class, anonymous inner class);
    - String handling (immutability, StringBuilder, StringBuffer, charAt, substring, indexOf, split);
    - Exception handling (try-catch-finally, throws, throw, custom exception);
    - Collections framework (List, Set, Map, Iterator);
    - I/O streams (File, InputStream, OutputStream, Reader, Writer, BufferedReader, BufferedWriter).

3. **Advanced stage: advanced syntax programming learning**.
    - Generics (generic class, generic method, wildcards);
    - Lambda expressions (functional interface, consumer, function, predicate);
    - Stream API (one-dimensional array, multi-dimensional array, array iteration);
    - Reflection;
    - Multithreading (thread, runnable, synchronized, lock, executor).

    Users need to select topics from the list above to generate the required source code.

4.2.3. Code Relevance to the Topic

Code relevance refers to the correlation between the AI-generated code and the user-specified topic.

At different stages, the requirements for the code relevance vary due to different teaching objectives.

In the primary stage, the requirement of code relevance is to help students grasp the fundamentals of *Java* syntax and lay a solid foundation for further learning. The containing content should focus on the current learning topic, and the generated code should closely align with the learning objectives, avoiding the premature introduction of complex logic or data structures. This ensures an engaging learning experience while minimizing frustration.

In the intermediate stage, the requirement will focus on mastering object-oriented concepts and developing API application skills. At this stage, the teaching content can introduce cross-chapter knowledge points to deepen understanding and connections. The generated source code should emphasize object-oriented principles, incorporate API usage, and appropriately review fundamental concepts.

In the advanced stage, students need to integrate various programming concepts, develop programming and system design skills, and understand advanced *Java* features. The teaching content can be more challenging, and the generated code should demonstrate a certain level of complexity and comprehensiveness.

For beginners, the learning material should avoid intermediate or advanced content. Similarly, intermediate learners should not be exposed to advanced content.

In this study, we used the *JavaParser* framework [36] to count the number of syntax elements appearing in the source code and calculate the *relevance index* between the code and the selected topic. Although the intermediate and advanced stages of learning involve previously learned topics, we avoided the previous topics to overshadow the main content. We assigned weights of 1, 2, and 3 to primary, intermediate, and advanced knowledge points, respectively, and used the following formula to calculate the code relevance:

$$R = \frac{\sum(N_1 \times w_1)}{\sum(N_2 \times w_2)} \tag{2}$$

where

- N_1 represents the number of appearances of the selected syntax elements;
- N_2 represents the number of appearances of all syntax elements;
- w_1 and w_2 represent the weights of the selected syntax elements and all syntax elements, respectively.

The $f_{\text{relevance}}$ score is determined by this code relevance R using the following formula:

$$f_{\text{relevance}} = \max\left(0, R - \lambda \cdot \frac{N_{\text{penalty}}}{N_{\text{total}}}\right) \tag{3}$$

where

- $R = \frac{\sum(N_1 \times w_1)}{\sum(N_2 \times w_2)}$ is the original relevance index;
- $N_{\text{penalty}}$is the number of syntax elements that should not appear at the current learning stage;
- N_{total} is the total number of syntax elements in the code;
- $\lambda \in [0, 1]$ is the penalty weight, controlling the influence of inappropriate content. Here, we set $\lambda = 0.5$ for the primary stage and $\lambda = 0.3$ for the intermediate stage.

In practical applications, to ensure that teaching objectives are met and based on our previous findings, the relevance of current topic should not fall below 50% in any learning stage.

4.2.4. Code Difficulty

Code difficulty refers to the complexity of a source code. Our previous study on the *JPLAS recommendation function* in [31] presents a way of dividing the difficulty weight according to syntax topics and calculating it. First, we assign the weights to the syntax topics based on their difficulty in *Java* learning:

1. **Syntax Topics with Weight 1:**
    - Variable;
    - Access Modifier;
    - Primitive Data Type;
    - Wrapper Class;
    - Operator;
    - Control Statement;
    - Array;
    - Common Word;
    - Code Block.
2. **Syntax Topics with Weight 2:**
    - String Functions;
    - Exception;
    - Package;
    - I/O.
3. **Syntax Topics with Weight 3:**
    - Class;
    - Interface;
    - Regular Expression;
    - Recursion;
    - Collections Framework.

Then, we extract the keywords of different syntax topics from the source code to calculate the summation of the weights. The same keyword is counted only once, even if it appears multiple times, and the calculated weight is the difficulty level of the code.

Building on our previous work, we evaluated the code difficulty based on a combination of factors, including the difficulty level of the syntax elements, the number of methods and classes, the code length, and the depth of inheritance and nested structures.

The requirements for *code difficulty* vary across different learning stages due to differing teaching objectives.

For the primary and intermediate stages, the generated code should be as simple as possible to foster interest and reduce frustration.

For the advanced stage, the difficulty of the generated code can be appropriately increased to challenge students and enhance their learning experience.

We use the following formula to calculate the *code difficulty*:

$$D = w_1 \sum(w_s \cdot S) + w_2 M + w_3 L + w_4 (I + N) \tag{4}$$

where

- D represents the overall code difficulty.
- S represents the number of syntax elements.
- w_s represents the weight of syntax elements.
- M represents the number of methods.
- L represents the total lines of code.
- I represents the depth of inheritance.
- N represents the number of nested structures.
- w_1, w_2, w_3, w_4 represent the weights assigned to each factor.

Additionally, the weights satisfy the following constraint:

$$w_1 + w_2 + w_3 + w_4 = 1 \tag{5}$$

For different learning stages, the weight distribution is adjusted accordingly:

- **Primary stage:** we set $w_1 = 0.6$, $w_2 = 0.15$, $w_3 = 0.15$, and $w_4 = 0.1$ to emphasize basic syntax comprehension while minimizing structural complexity.
- **Intermediate stage:** we set $w_1 = 0.4$, $w_2 = 0.2$, $w_3 = 0.2$, and $w_4 = 0.2$ to balance syntax, method usage, code length, and structural complexity as students build deeper understanding.
- **Advanced stage:** we set $w_1 = 0.25$, $w_2 = 0.2$, $w_3 = 0.15$, and $w_4 = 0.4$ to place greater emphasis on structural complexity and advanced programming concepts.
- **Rationale for weight selection:** The weight distribution was determined based on an analysis of standard teaching materials and historical student learning performance. We referred to widely used introductory programming textbooks and teaching materials to identify which elements are most emphasized at different stages of learning. Additionally, we reviewed past records of student progress and performance to align the weights with the actual learning challenges encountered at different proficiency levels. While the weighting process inherently involved some subjectivity, it was grounded in established educational practices and designed to align with the cognitive progression of novice to advanced learners.

The standard difficulty value D_{std} and the deviation tolerance value D_{tol} for a given topic are computed using the D formula based on a set of high-quality benchmark code for that topic.

$$D_{\text{std}} = \frac{1}{n} \sum_{i=1}^{n} D_i \tag{6}$$

$$D_{\text{tol}} = \frac{1}{n} \sum_{i=1}^{n} (D_i - D_{\text{std}})^2 \tag{7}$$

The $f_{\text{difficulty}}$ scoreis then determined by the deviation between the actual code difficulty D and the standard difficulty level D_{std}, as well as the deviation tolerance D_{tol} of the topic, using the following formula:

$$f_{\text{difficulty}} = \exp\left(-\frac{|D - D_{\text{std}}|}{D_{\text{tol}}}\right) \tag{8}$$

Here, D is the difficulty value of the generated code. The function ensures that the scores fall within the interval $(0, 1]$, rewarding code that closely matches the target difficulty and penalizing significant deviations.

In practical applications, it is essential to ensure that the generated code strictly adheres to the scope of the current learning stage. Any content that exceeds the complexity or concepts intended for a particular stage may overwhelm learners and hinder their process. Therefore, no content should exceed the scope of the current learning stage at any time.

4.2.5. Feasibility of Problem Generation

Problem generation feasibility evaluates whether the generated code can be used for problem creation through the *blank element selection algorithm*. An excellent code should not only be accurate but also provide more opportunities for problem generation.

The $f_{\text{feasibility}}$ score is determined by the ratio of the number of blank elements in the source code to the total number of tokens in the code using the following formula:

$$f_{\text{feasibility}} = \frac{B}{T} \tag{9}$$

- B represents the number of blank elements selected for problem generation.
- T represents the total number of tokens in the source code.

4.2.6. Objective Function

Based on the above four evaluation criteria, we propose the following objective function F to evaluate the quality of the generated source code for *Java* programming learning:

$$F = 100 \times \left(w_a \cdot f_{\text{accuracy}} + w_f \cdot f_{\text{feasibility}} + w_r \cdot f_{\text{relevance}} + w_d \cdot f_{\text{difficulty}} \right) \tag{10}$$

where

- w_a represents the weight for accuracy;
- w_f represents the weight for feasibility;
- w_r represents the weight for relevance;
- w_d represents the weight for difficulty.

The values of w_a, w_f, w_r, and w_d can vary depending on the learning topic. In the initial stage, we set the values of both w_a, w_f, w_r, and w_d to 0.25. During actual system usage, users can adjust these weights according to their specific instructional needs and learning objectives.

4.3. Generative AI

The history of *generative AI* can be traced back to early artificial neural networks, with key milestones including the introduction of *artificial neurons* in the 1950s and the *back-propagation* algorithm in the 1980s [37]. In 2017, the introduction of the *transformer* architecture by Vaswani et al. [38] laid the foundation for modern large-scale models. Since then, *LLMs*, represented by *ChatGPT*, have demonstrated remarkable natural language generation capabilities. Recently, research and products applying *LLMs* into the field of education have emerged rapidly. *Generative AI* is continuously evolving, offering groundbreaking possibilities for AI-driven content creation and automation.

AI Model Introduction

In this study, we selected three representative AI models for experiments. Here, we provide a brief overview of their performance.

ChatGPT-4o is *OpenAI's* latest model, performing the best performance on general tasks, in reasoning abilities, and in multimodal support. However, it is closed-source and only available through *OpenAI's* API and *ChatGPT* platform.

DeepSeek-R1-7B is an open-source transformer model that balances performance and efficiency, making it suitable for local deployment with consumer-level GPUs.

LlaMA3.2-1B is a lightweight, open-source model, making it ideal for low-power devices and edge computing. It is designed to be fast and efficient, and can run on a PC without GPUs.

4.4. Prompt Engineering

Prompt engineering refers to the optimization of input prompts to maximize the quality and accuracy of AI-generated outputs. It is a key technique for enhancing AI-generated content and is widely used in natural language processing, question answering, code generations, text summarization, and other fields.

Due to device or network limitations, it is not always possible to use the optimal AI model. Therefore, adopting suitable *prompt strategies* to obtain the required output has become a more economically viable approach.

Based on the working principles of and practical experience with *LLMs*, *prompt engineering* includes the following core concepts:

1. **Clarity:** prompts should be clear, specific, and avoid ambiguity.
2. **Context:** providing relevant background information improves accuracy.
3. **Instruction-based:** the model is directly instructed on what to do, such as "list the steps" or "summarize briefly."
4. **Examples (few-shot learning):** providing examples helps guide AI to generate responses in the expected format.
5. **Constraints:** setting limits on word count, format, or style ensures the output meets specific requirements.
6. **Temperature and top-p:** the *temperature* parameter controls the randomness of the output—lower values result in more deterministic responses, and, the *top-p* parameter governs nucleus sampling, a technique that introduces controlled randomness to the model's output.

Based on the above core concepts and extensive practical experience, the following are the common prompt engineering techniques:

1. **Zero-shot and few-shot:** *Zero-shot* and *few-shot* are two key prompting techniques used in *LLMs* to control how they generate responses. *Zero-shot* means that the model is given a prompt without any example and is expected to generate a correct response based on its pre-trained knowledge. It relies entirely on the AI's pre-existing understanding of language and concepts. *Few-shot*, on the other hand, means that the model is provided with a few examples in the prompt before being asked to generate a response. The examples help the model understand the pattern and produce more accurate results.

   Example:

   - *Zero-shot:* "What is the capital of France?" The model will respond, "Paris."
   - *Few-shot:* "The capital of Italy is Rome. The capital of Germany is Berlin. What is the capital of France?" The model will respond, "Paris."

2. **Chain-of-thought (CoT):** *CoT prompting* is a technique used in *LLMs* to improve reasoning and problem solving by encouraging the model to break down its thought process step by step. Instead of directly generating an answer, the model explains its

reasoning before reaching a conclusion. By mimicking human thought processes, *CoT prompting* helps AI produce more accurate and explainable responses.

Example:

When asked, "What is 37 multiplied by 6?", the model might respond with: "First, I multiply 30 by 6, which gives me 180. Then, I multiply 7 by 6, which gives me 42. Finally, I add 180 and 42, which gives me 222."

This step-by-step breakdown demonstrates reasoning before arriving at the final answer.

3. **Black-box prompt optimization (BPO):** *BPO* is a technique for improving prompts used in *LLMs* without knowing the internal workings of the model. Since the *LLM* is treated as a "black box", the optimization is achieved by evaluating different prompts based on output performance rather than analyzing the model's structure or gradients. It presents a practical way of refining AI interactions without accessing the model's internal mechanisms.

   Example:

   - *Initial prompt:* "Tell me the benefits of regular exercise."
   - *Revised prompt:* "What are the top three benefits of daily physical activity for mental health?"

   The model's response to the initial prompt might not fully meet the expectations in terms of detail or relevance. By evaluating the response, the prompt is refined to focus more specifically on the mental health benefits. This optimization continues iteratively based on the quality of the generated responses to each version of the prompt.

5. Experiments

In this study, we conducted extensive experiments to evaluate the performance of three *prompt engineering* methods with three *AI models* for generating proper source codes for *Java* programming learning.

5.1. Prompt Strategies

To avoid overfitting and reduce human bias, we did not provide sample code in the prompt. Below is the prompt used for the experiment, taking the "For Loop" topic as an example.

- **Zero-shot:**

  Input:
  Please help me generate a piece of code using java, the subject is: for loop.

- **Chain-of-thought (CoT):**

  Input:
  I need to generate a piece of Java code related to the topic: for loop. Before generating the code, let us break down the process step by step:

  1. Understanding the topic:
     - The subject is "For Loop", so the generated code must showcase a for loop in a meaningful way.
     - The code should be correct and syntactically valid in Java.
  2. Evaluating difficulty and structure:
     - The code should not be too simple (e.g., just printing numbers in a loop) but also not too complex.

- It should demonstrate a common programming pattern using a for loop.

3. Ensuring questionability:
   - The code should be suitable for generating programming exercises.
   - It should have elements that can be modified or extended for students to practice.
4. Code generation:
   - Now, generate a Java program that meets these requirements.
   - Ensure the code compiles correctly.

- **Black-box prompt optimization (BPO):**
  Figure 1 shows a flow chart of BPO processing.

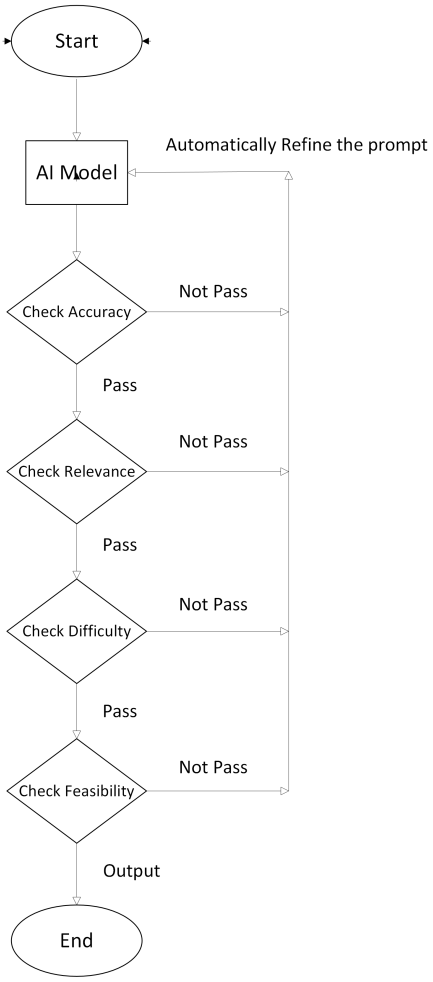

Figure 1. Flow chart of *BPO* processing.

Input:
Your task is to generate a Java program related to the topic: for loop.

After generating the code, the program calls the *objective function F* to evaluate it. It systematically checks whether the code meets the four evaluation criteria. If any criteria is not met, the program provides feedback along with the previously generated code, guiding the AI to refine and improve it.

Below are some example prompts for guiding the AI in making targeted modifications.

Possible Input:
The above is the *for Llop* code you just generated. I noticed that the class name does not follow the required naming convention. Please correct it.

Possible Input:
The above is the *for loop* code you just generated. I noticed that it is not well-uited for fill-in-the-blank questions. Please revise it accordingly.

Possible Input:
The above is the *for loop* code you just generated. I think the difficulty is a bit too high for beginners. Please simplify it.

Possible Input:
The above is the *for loop* code you just generated. I think it contains some advanced concepts beyond the intended scope. Please remove them.

Possible Input:
The above is the *for loop* code you just generated. I think it is not closely aligned with the topic. Please revise it to better fit the theme.

5.2. Experiment Design

5.2.1. Objective

We aimed to compare the performance of different prompting strategies across various AI models (*ChatGPT-4o*, *DeepSeek-R1-7B*, *Llama3.2-1B*) and different difficulty levels of *Java* learning topics (primary, intermediate, and advanced). Additionally, we sought to determine the optimal number of *BPO* calls required to achieve a relatively high score under different conditions, providing guidance for system deployment and real-world application.

5.2.2. Variables

The experiment involved the following variables:

- **AI Models**: *ChatGPT-4o*, *DeepSeek-R1-7B*, *Llama3.2-1B*.
- **Prompt strategies**:
  - **Zero-shot**: direct generation without additional reasoning or optimization.
  - **CoT (chain of thought)**: applying step-by-step reasoning during generation.
  - **BPO (black-box prompt optimization)**: performing up to 3 optimization calls per generation.
  - **CoT + BPO**: first applying CoT reasoning, then optimizing with BPO (maximum 3 calls).
- **Difficulty levels**: primary, intermediate, advanced.

5.2.3. Procedure

1. For each combination of an AI model, a prompt strategy, and a topic difficulty level, generate outputs.
2. For BPO-related strategies, continue optimization until either
   - The score does not improve compared to the previous attempt; or
   - The maximum limit of 3 calls is reached.
3. Repeat each experimental condition 20 times independently to ensure statistical significance.

5.2.4. Evaluation Metrics

Each generated output was evaluated using the objection function in Section 4.

5.2.5. Statistical Analysis

After completing the 20 repetitions for each condition:

- Calculate the average score and the average *BPO* call times across the 20 trials.
- Use the average score and the average *BPO* call times as the final performance indicator for comparison among different strategies and models.

5.3. Result and Analysis

As the most advanced commercial AI model, *ChatGPT-4o* exhibits strong performance across all Java topic levels. As shown in Table 1, in primary and intermediate tasks, all strategies (Zero-shot, CoT, BPO, CoT+BPO) achieve relatively high scores, and the gap between strategies remains small. This reflects *ChatGPT-4o*'s inherently strong reasoning and generalization ability.

Table 1. Performance comparison for *ChatGPT-4o*.

Prompt Strategy	Difficulty	Avg.Score	BPO Calls
Zero-shot	Primary	79.6	-
Zero-shot	Intermediate	75.1	-
Zero-shot	Advanced	65.6	-
CoT	Primary	79.4	-
CoT	Intermediate	78.6	-
CoT	Advanced	66.3	-
BPO	Primary	83.8	0.8
BPO	Intermediate	79.5	1.5
BPO	Advanced	71.9	2.5
CoT+BPO	Primary	82.3	0.6
CoT+BPO	Intermediate	81.4	1.6
CoT+BPO	Advanced	73.5	2.3

However, in advanced tasks, a notable performance drop is observed for zero-shot (65.6) and CoT (66.3), while BPO-based strategies push the scores above 70. This suggests that when the task complexity increases, prompt optimization becomes necessary even for top-tier models.

Moreover, *CoT+BPO* achieves higher first-generation quality (fewer BPO rounds required) compared to pure BPO. This indicates that explicitly guiding the model's reasoning before optimization is more efficient for complex tasks.

As shown in Table 2, *DeepSeek-R1-7B* demonstrates strong competence on primary tasks, with zero-shot achieving over 73 points. However, as task difficulty increases, the performance degrades significantly under basic strategies, indicating its limitations in complex reasoning.

Compared to ChatGPT-4o, the benefit of BPO and CoT+BPO is more pronounced in *DeepSeek-R1-7B*, especially at the intermediate and advanced levels. CoT+BPO consistently outperforms thr other strategies, suggesting that combining structured reasoning (CoT) with optimization (BPO) can compensate for the model's native reasoning deficiencies.

Furthermore, the number of BPO calls for *DeepSeek-R1-7B* tends to be higher than *ChatGPT-4o*, reflecting the need for more refinement cycles to achieve acceptable output.

As shown in Table 3, the lightweight *Llama3.2-1B* model presents a more challenging case. Its baseline performance under zero-shot drops sharply as task complexity increases, especially at the advanced level (only 44.3 points).

Table 2. Performance comparison for *DeepSeek-R1-7B*.

Prompt Strategy	Difficulty	Avg.Score	BPO Calls
Zero-shot	Primary	73.3	-
Zero-shot	Intermediate	63.7	-
Zero-shot	Advanced	62.1	-
CoT	Primary	75.6	-
CoT	Intermediate	64.2	-
CoT	Advanced	63.9	-
BPO	Primary	76.1	1.8
BPO	Intermediate	66.3	1.9
BPO	Advanced	64.7	3
CoT+BPO	Primary	74.9	1.5
CoT+BPO	Intermediate	67.6	1.6
CoT+BPO	Advanced	65.8	2.9

Table 3. Performance comparison for *Llama3.2-1B*.

Prompt Strategy	Difficulty	Avg.Score	BPO Calls
Zero-shot	Primary	70.7	-
Zero-shot	Intermediate	57.4	-
Zero-shot	Advanced	44.3	-
CoT	Primary	71.5	-
CoT	Intermediate	59.5	-
CoT	Advanced	51.8	-
BPO	Primary	72.2	2.2
BPO	Intermediate	61.5	3
BPO	Advanced	55.8	3
CoT+BPO	Primary	71.3	1.9
CoT+BPO	Intermediate	61.8	3
CoT+BPO	Advanced	58.3	3

Both BPO and CoT+BPO show clear improvement, but the absolute scores remain lower than higher-parameter models. Importantly, BPO calls reach the maximum limit (three) in almost all intermediate and advanced tasks, indicating the model struggles to refine its outputs without extensive optimization.

Overall, BPO and CoT+BPO are critical for enabling small models to perform acceptably, but they cannot fully bridge the gap in the performance of larger models.

As shown in Table 4, allowing unlimited BPO iterations highlights the limitations of small models. For *Llama3.2-1B*, even after multiple optimization rounds (up to 6.7 iterations for advanced tasks), the final scores remain lower than larger models.

Table 4. Performance of infinite *BPO* across AI models.

AI model	Task Difficulty	Score and Iterations
ChatGPT-4o	Primary	Score: 82.6, Iterations: 0.7
	Intermediate	Score: 79.3, Iterations: 1.5
	Advanced	Score: 72.5, Iterations: 2.3
DeepSeek-R1-7B	Primary	Score: 75.8, Iterations: 1.6
	Intermediate	Score: 66.7, Iterations: 1.8
	Advanced	Score: 66.9, Iterations: 3.8
Llama3.2-1B	Primary	Score: 71.2, Iterations: 1.9
	Intermediate	Score: 65.3, Iterations: 4.3
	Advanced	Score: 62.8, Iterations: 6.7

This shows that while BPO can substantially improve small model outputs, its effect diminishes beyond a certain point. In contrast, large models like *ChatGPT-4o* achieve high scores with minimal optimization, suggesting that BPO is more cost-effective for large models and becomes increasingly expensive and less effective for smaller ones.

Summary

The experimental results demonstrate that

- For top-tier models (e.g., *ChatGPT-4o*), prompt strategy has a limited influence on simple tasks but becomes critical in complex tasks.
- For mid-range models (e.g., *DeepSeek-R1-7B*), combining CoT and BPO yields the best performance across all difficulty levels.
- For lightweight models (e.g., *Llama3.2-1B*), BPO is necessary but insufficient to fully close the performance gap.
- BPO iterations should be carefully managed: unlimited iterations can significantly increase computation cost with diminishing returns, especially for small models.

These findings provide practical insights into selecting appropriate prompting strategies and models for different deployment scenarios.

For environments with limited computational resources, mid-range models with CoT+BPO are strong candidates, while high-performance models like *ChatGPT-4o* are ideal for applications demanding top-tier performance across a wide range of task difficulties. Lightweight models like *Llama3.2-1B* can still serve in specific scenarios when the computational resources are limited or only primary stage tasks are required.

5.4. Student Participation Testing and Evaluation

To further evaluate the practical effectiveness of this system, we conducted a student participation test involving eight participants (four with prior *Java* learning experience and four without).

Participants were asked to complete exercises based on two types of problems:

- EFP with source code manually selected by instructors;
- EFP with source code generated by different AI models with the highest scores from previous experiments.

Importantly, the participants were not informed of the origin of the exercises. The four participants with *Java* learning experience completed the intermediate-difficulty exercises, while the other four without prior learning experience completed the primary-difficulty exercises. Half of the participants worked on AI-generated exercises, while the other half worked on manually selected exercises.

After completing the exercises, the students were asked to evaluate each exercise on four aspects—difficulty, correctness, topic relevance, and helpfulness to their learning—using a five-star rating system.

To enhance the statistical credibility of the findings despite the limited sample size, we calculated the 95% confidence intervals (as shown in brackets) and effect sizes (Cohen's d) for each evaluation aspect. The results are summarized as follows:

- **Difficulty**: manually selected exercises were rated slightly higher (Cohen's $d = -1.22$), indicating a potentially noticeable difference in perceived difficulty.
- **Correctness**: both groups assigned identical average scores with overlapping confidence intervals, suggesting no meaningful difference (Cohen's $d = 0.00$).
- **Topic relevance**: AI-generated exercises performed slightly better (Cohen's $d = 0.71$), reflecting a moderate positive effect.

- **Helpfulness**: manually selected exercises had a moderate advantage (Cohen's $d = -0.71$).

As shown in Table 5, although individual differences existed, the average scores across all four evaluation aspects (difficulty, correctness, topic relevance, and helpfulness) were comparable between AI-generated and manually selected exercises. The AI-generated exercises demonstrated quality and educational effectiveness similar to those of manually selected ones.

Table 5. Student evaluation results (average scores out of 5, with 95% confidence intervals).

Evaluation Aspect	AI-Generated Exercises	Manually Selected Exercises
Difficulty	4.00 [4.00, 4.00]	4.25 [3.90, 4.60]
Correctness	4.75 [4.40, 5.10]	4.75 [4.40, 5.10]
Topic Relevance	4.50 [4.00, 5.00]	4.25 [3.90, 4.60]
Helpfulness	4.00 [3.50, 4.50]	4.25 [3.90, 4.60]

5.5. Functional Testing and Evaluation

To validate the effectiveness and efficiency of this system, we conducted a series of functionality tests focusing on different aspects of the system performance on a PC with an *i5-11400H CPU*, *16 GB of RAM*, and *Llama3.2-1B*. Since this system was primarily designed for beginners, we referred to the book *Java: A Beginner's Guide* and the website *W3Schools* and selected 10 learning topics suitable for beginners to conduct the tests.

As shown in Table 6, the overall execution time and results were within an acceptable range.

Table 6. Functionality test.

Java Topic	CPU Time (s)	Lines of Code	BPO Iterations	EFP Blanks	Difficulty Score
Variable	2.2	16	0	5	19
Control Statement	3.4	22	1	8	32
Class	4.3	35	2	7	35
Exception Handling	5.8	40	2	9	60
Operators	2.8	32	0	7	40
Collections Framework	6.1	37	2	10	56
I/O Operations	5.5	20	2	6	42
Arrays	3.8	22	1	5	30
String Manipulation	4.5	30	2	8	48
Interface	5.1	44	2	13	70

The results demonstrate that this system effectively adapts to different *Java* learning topics, ensuring the performance, correctness, and feasibility for EFP instance generations.

6. Application to Element Fill-in-the-Blank Problem Creation

In this section, we present an application of the optimal prompt strategy of generating source codes for *Java* programming learning that was derived from experiments. As an application, we implemented a web-based system that automatically createsda proper element fill-in-the-blank problem (EFP) instance in *JPLAS* without manually selecting the source code.

6.1. Adopted Open Source Software

For this web application system, we adopted the following open-source software:

1. **Spring Boot:** *Spring Boot* is an open-source framework used to simplify the development of *Java*-based applications. It provides a set of conventions and tools for building production-ready, stand-alone, and microservice-based applications. *Spring*

Boot allows developers to focus on business logic, while it handles the setup, configuration, and dependencies of the application. It includes embedded servers like *Tomcat*, which means developers can run *Spring Boot* applications directly without needing external servers.

2. **jQuery:** *jQuery* is a fast, lightweight, and feature-rich *JavaScript* library. It simplifies *HTML* document traversal and manipulation, event handling, and animation, making it easier to work with *JavaScript*. *jQuery* provides an intuitive syntax for tasks like *DOM* manipulation, *Ajax* requests, and cross-browser compatibility. Widely used in web developments, *jQuery* allows developers to create interactive and dynamic websites quickly and efficiently.
3. **Ollama:** *Ollama* is a platform designed for deploying and running *LLMs* locally on personal computers. It enables developers to utilize *LLMs* in a wide range of applications without relying on cloud-based solutions, ensuring better data privacy and control over the models. *Ollama* supports various models, including *Llama3*, and allows for integration with AI-driven systems in multiple industries, including education, healthcare, and more.
4. **Docker:** *Docker* is a platform that allows developers to package applications and their dependencies into containers, ensuring that the application works seamlessly across different environments. Containers are lightweight, portable, and consistent, which makes deploying and managing applications much easier. *Docker* simplifies software distribution, improves scalability, and provides isolation, making it an essential tool for modern *DevOps* practices and microservice architectures.

We chose *Spring Boot*, *jQuery*, *Ollama*, and *Docker* as the technology stack for this system. *Spring Boot* provided an efficient development experience and simplified configuration management, making it ideal for building and maintaining backend services that need to interact with AI models and the frontend. While modern frameworks like *React* [39] and *Vue* [40] were more popular at the time, we chose *jQuery* due to its use in our previous system. This approach avoided changes to existing functionality and ensured system stability and compatibility. Additionally, *jQuery*'s simplicity and broad support helped quickly implement dynamic pages and handle backend interactions. *Ollama* allowed us to run LLMs locally, reducing reliance on cloud-based services and enhancing data privacy and control, which are especially valuable for applications in the education field. Finally, *Docker*, as a containerization technology, helped us package the system and its dependencies into containers, simplifying deployment, improving maintainability and scalability, and enabling code reuse across different environments.

6.2. Software Architecture

Figure 2 presents an overview of our system, which consists of four key components:

- **Frontend (*jQuery*)** handles the user interactions, such as selecting a topic, modifying an AI-generated code, and managing problems.
- **Backend (*Spring Boot*)** manages API requests, processes user inputs, communicates with the AI model, and provides the necessary logic for evaluations and refinements.
- **AI component (*Ollama*)** generates a *Java* source code based on the selected topic and refines it according to user instructions.
- **Containerization (*Docker*)** packages the entire system into a *Docker* container, allowing for the easy deployment and installation with a single command across different environments.

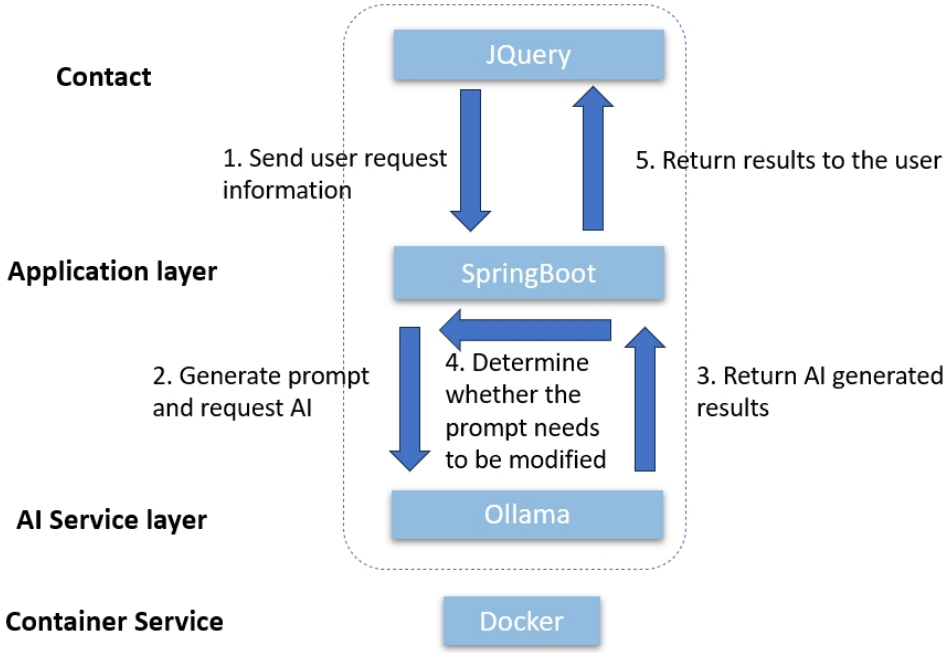

Figure 2. EFP creation system overview.

6.3. Functional Overview

Our system provides four core functions: Figure 3 presents the user interface for setting parameters. Figure 4 presents the user interface for generating code. Figure 5 presents the user interface for modifying questions. Figure 6 presents the user interface for management.

1. Code generation based on topic selection:
   - The user selects a *Java* programming topic.
   - The back end generates the optimal prompt for the selected topic and calls *Ollama AI* to generate a *Java* code.
2. AI-guided code modification:
   - The user can provide specific modification requirements, such as simplifying the code or aligning it better with a given topic.
   - The system sends the existing code along with the modification instructions to *Ollama*, which generates a refined version.
   - This iterative process helps improve the quality and suitability of the generated code.
3. Automated question generation:
   - Once a valid *Java* code is generated, the system automatically creates an EFP instance using the *blank element selection algorithm*.
4. Problem downloading:
   - The user can download the generated source code and EFP instance for future use.

Figure 7 presents one of generated source code.

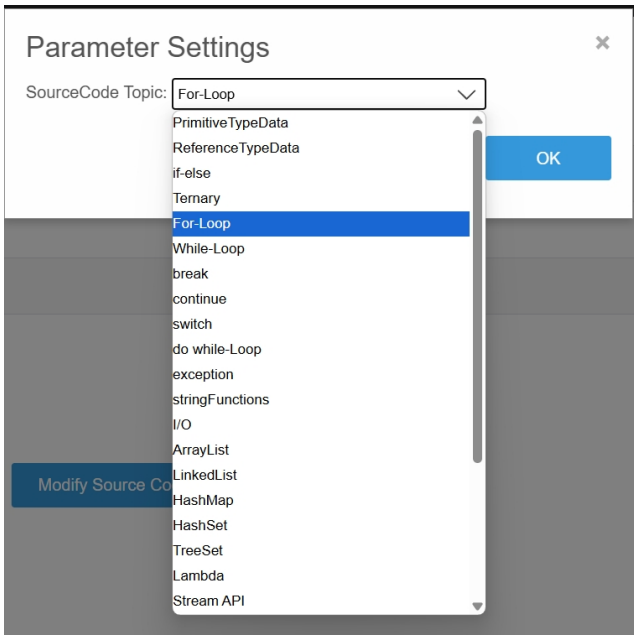

Figure 3. User interface for parameter setting.

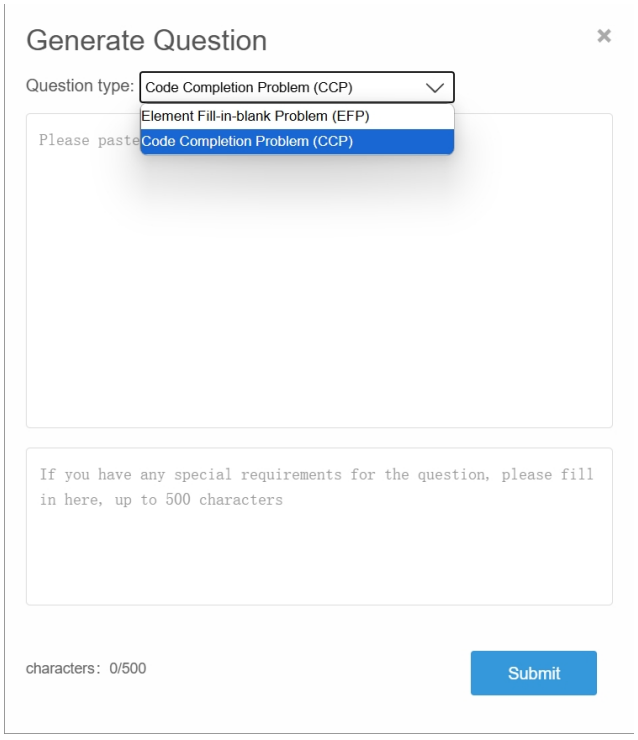

Figure 4. User interface for generating code.

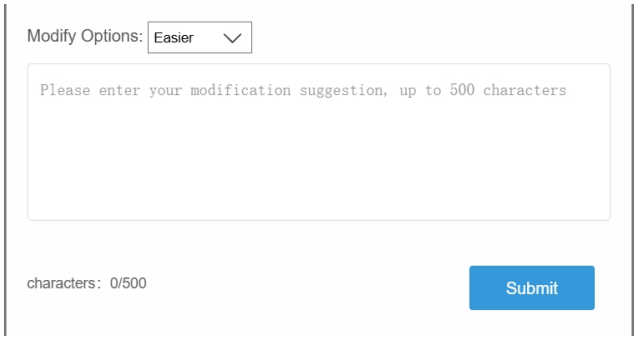

Figure 5. User interface for modifying questions.

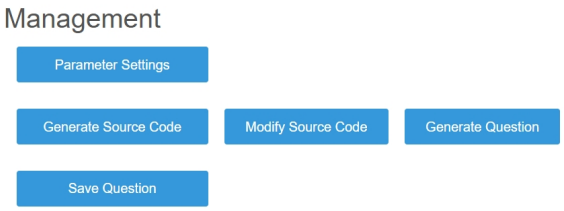

Figure 6. User interface for management.

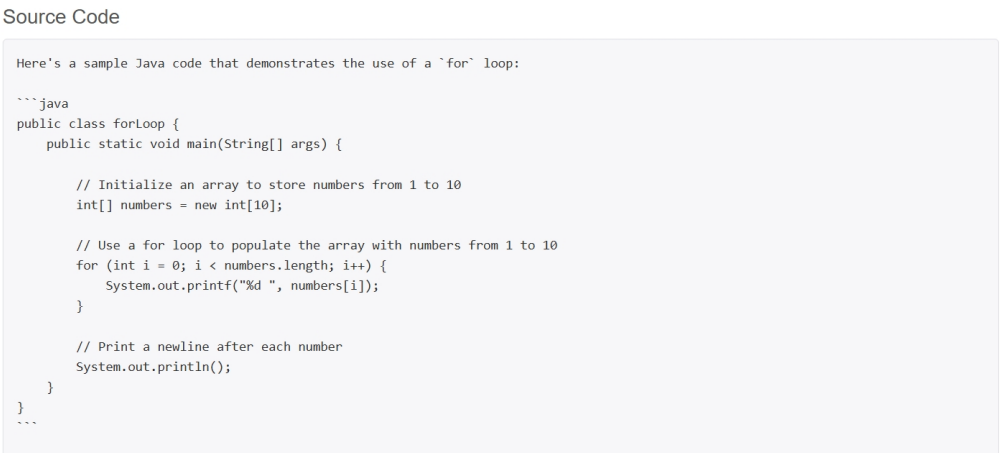

Figure 7. Generated source code.

7. Conclusions

This paper experimentally analyzed the effectiveness of generating Java source codes for learning using three prompt strategies and AI models as well as identified the optimal prompt strategy for different tasks. The prompt strategy was integrated into the system, ensuring code quality while reducing the burden on teachers in preparing teaching materials. Prompt engineering does not require fine-tuning or additional training of the model. Instead, it significantly improves the quality of the generated code by simply optimizing the prompt text, thereby reducing the cost of optimizing the AI outputs. Moreover, this approach is not limited to *Java* code generation but can also be extended to other languages, providing an efficient optimization method for AI applications in programming education.

Despite the results achieved in this study, there are still some limitations. First, this prompt strategy was only proven effective for a specific version of the AI model. Its effectiveness may not hold if the model is updated. Second, this study focused only on *Java* programming tasks, and its applicability to other programming languages or more complex software engineering tasks remains unverified. Additionally, the evaluation primarily relied on experimental data, lacking long-term validations in large-scale real-world classroom settings.

Our research can be expanded in the following directions:

1. **Cross-model adaptability:** we will investigate the applicability of the optimal prompt strategy across different AI models to develop a more general and robust optimization method.
2. **Multi-language code generation:** we will explore the effectiveness of this approach in other programming languages, such as *Python* and *C++*, to further validate its applicability.
3. **Large-scale user testing:** we will deploy the system in real educational settings, collect student usage data, and analyze the impacts of different prompt strategies on learning outcomes to further refine AI-assisted programming education.

In summary, this study provides a low-cost and efficient optimization strategy for AI applications in programming education. Future work will focus on expanding its applicability and refining it based on real-world teaching scenarios.

Author Contributions: Methodology, Z.Z.; software, Z.Z.; investigation, M.M. and S.T.A.; data curation, M.M. and S.T.A.; writing—original draft, Z.Z.; writing—review and editing, Z.Z. and N.F.; supervision, W.-C.K. and Y.-F.L.; project administration, N.F. All authors have read and agreed to the published version of this manuscript.

Funding: This research received no external funding.

Data Availability Statement: The original contributions presented in this study are included in this article. Further inquiries can be directed to the corresponding author.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Netguru. Is Java Still Used? Current Trends and Market Demand in 2025. Available online: https://www.netguru.com/blog/is-java-still-used-in-2025 (accessed on 18 February 2025).
2. Ishihara, N.; Funabiki, N.; Kuribayashi, M.; Kao, W.-C. A software architecture for Java programming learning assistant system. *Int. J. Comput. Softw. Eng.* **2017**, *2*, 116. [CrossRef] [PubMed]
3. Aung, S.T.; Funabiki, N.; Syaifudin, Y.W.; Kyaw, H.H.S.; Aung, S.L.; Dim, N.K.; Kao, W.-C. A Proposal of Grammar-Concept Understanding Problem in Java Programming Learning Assistant System. *J. Adv. Inf. Technol.* **2021**, *12*, 342–350.
4. Funabiki, N.; Zaw, K.K.; Ishihara, N.; Kao, W.C. A Graph-Based Blank Element Selection Algorithm for Fill-in-Blank Problems in Java Programming Learning Assistant System. *IAENG Int. J. Comput. Sci.* **2017**, *44*, 247–260.
5. OpenAI. ChatGPT. Available online: https://openai.com/index/chatgpt/ (accessed on 15 March 2025).
6. Chen, B.; Zhang, Z.; Langrené, N.; Zhu, S. Unleashing the Potential of Prompt Engineering in Large Language Models: A Comprehensive Review. *arXiv* **2023**, arXiv:2310.14735.
7. OpenAI. GPT-4o: OpenAI's Newest Model. Available online: https://openai.com/index/hello-gpt-4o/ (accessed on 15 March 2025).
8. DeepSeek. DeepSeek-R1 Model on Ollama. Available online: https://ollama.com/library/deepseek-r1 (accessed on 15 March 2025).
9. Meta. LLaMA 3.2 Model on Ollama. Available online: https://ollama.com/library/llama3.2 (accessed on 15 March 2025).
10. Spring. Spring Boot. Available online: https://spring.io/projects/spring-boot (accessed on 15 March 2025).
11. jQuery. jQuery: The Write Less, Do More, JavaScript Library. Available online: https://jquery.com/ (accessed on 15 March 2025).
12. Ollama. Ollama: Run AI Models Locally. Available online: https://ollama.com/ (accessed on 15 March 2025).
13. Docker. Docker: Empowering Developers to Build, Share, and Run Applications. Available online: https://www.docker.com/ (accessed on 15 March 2025).
14. McGill, T.; Volet, S. A Conceptual Framework for Analyzing Students' Knowledge of Programming. *J. Res. Comput. Educ.* **1997**, *29*, 276–297. [CrossRef]
15. Altadmri, A.; Brown, N.C. 37 Million Compilations: Investigating Novice Programming Mistakes in Large-Scale Student Data. In Proceedings of the 46th ACM Technical Symposium on Computer Science Education, Kansas City, MO, USA, 4–7 March 2015; pp. 522–527. [CrossRef]
16. Gomes, A.; Mendes, A.J. An Environment to Improve Programming Education. In Proceedings of the 2007 International Conference on Computer Systems and Technologies, Rousse, Bulgaria, 14–15 June 2007; Volume 88, pp. 1–6. [CrossRef]
17. Sorva, J.; Karavirta, V.; Malmi, L. A Review of Generic Program Visualization Systems for Introductory Programming Education. *Acm Trans. Comput. Educ.* **2013**, *13*, 15. [CrossRef]
18. Medeiros, R.P.; Ramalho, G.L.; Falcão, T.P. A Systematic Literature Review on Teaching and Learning Introductory Programming in Higher Education. *IEEE Trans. Educ.* **2019**, *62*, 77–90. [CrossRef]
19. Lindberg, R.S.; Laine, T.H.; Haaranen, L. Gamifying Programming Education in K–12: A Review of Programming Curricula in Seven Countries and Programming Games. *Br. J. Educ. Technol.* **2019**, *50*, 1979–1995. [CrossRef]
20. Olsson, M.; Mozelius, P.; Collin, J. Visualisation and Gamification of E-Learning and Programming Education. *Electron. J. E-Learn.* **2015**, *13*, 452–465.
21. Luckin, R.; Holmes, W. *Intelligence Unleashed: An Argument for AI in Education*; Pearson: London, UK, 2016.
22. Chen, L.; Chen, P.; Lin, Z. Artificial Intelligence in Education: A Review. *IEEE Access* **2020**, *8*, 75264–75278. [CrossRef]
23. Baidoo-Anu, D.; Ansah, L.O. Education in the Era of Generative Artificial Intelligence (AI): Understanding the Potential Benefits of ChatGPT in Promoting Teaching and Learning. *J. AI* **2023**, *7*, 52–62. [CrossRef]

24. Coding Rooms. Coding Rooms: Developer Training & Enablement Platform. Available online: https://www.codingrooms.com/ (accessed on 15 March 2025).
25. Khan Academy. Khan Academy: For Every Student, Every Classroom. Real Results. Available online: https://www.khanacademy.org/ (accessed on 15 March 2025).
26. Khanmigo. Khanmigo: An AI-Powered Tutor and Teaching Assistant. Available online: https://www.khanmigo.ai/ (accessed on 15 March 2025).
27. GitHub Copilot. GitHub Copilot: Your AI Pair Programmer. Available online: https://github.com/features/copilot (accessed on 15 March 2025).
28. Cheng, J.; Liu, X.; Zheng, K.; Ke, P.; Wang, H.; Dong, Y.; Huang, M. Black-Box Prompt Optimization: Aligning Large Language Models Without Model Training. *arXiv* **2023**, arXiv:2311.04155.
29. Agarwal, E.; Singh, J.; Dani, V.; Magazine, R.; Ganu, T.; Nambi, A. PromptWizard: Task-Aware Prompt Optimization Framework. *arXiv* **2024**, arXiv:2405.18369.
30. Fernando, C.; Banarse, D.; Michalewski, H.; Osindero, S.; Rocktäschel, T. PromptBreeder: Self-Referential Self-Improvement via Prompt Evolution. *arXiv* **2023**, arXiv:2309.16797.
31. Wint, S.S.; Funabiki, N. A proposal of recommendation function for element fill-in-Blank problems in Java programming learning assistant system. *Int. J. Web Inf. Syst.* **2021**, *17*, 140–152. [CrossRef]
32. JFlex. JFlex: A Lexical Analyzer Generator for Java. Available online: https://www.jflex.de/ (accessed on 18 February 2025).
33. JUnit. JUnit. Available online: https://github.com/junit-team/junit5/ (accessed on 18 February 2025).
34. EvoSuite. EvoSuite: Automated Test Suite Generation for Java. Available online: https://www.evosuite.org/ (accessed on 27 April 2025).
35. Niemeyer, P.; Knudsen, J. *Learning Java*; O'Reilly Media, Inc.: Newton, MA, USA, 2005.
36. JavaParser. JavaParser: The Most Popular Parser for the Java Language. Available online: https://javaparser.org (accessed on 18 February 2025).
37. Cao, Y.; Li, S.; Liu, Y.; Yan, Z.; Dai, Y.; Yu, P.S.; Sun, L. A comprehensive survey of AI-generated content (AIGC): A history of generative AI from GAN to ChatGPT. *arXiv* **2023**, arXiv:2303.04226.
38. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, Ł.; Polosukhin, I. Attention is all you need. *Adv. Neural Inf. Process. Syst.* **2017**, *30*, 5998–6008.
39. React. React. Available online: https://reactjs.org/ (accessed on 18 February 2025).
40. Vue. Vue.js. Available online: https://vuejs.org/ (accessed on 18 February 2025).