

## Research Article

# Proposal of Coding Support Methodology Using Artificial Language and Generative AI to Address the Shortage of Software Developers

Masaya Nakahara<sup>1\*</sup>, Kaito Andachi<sup>2</sup>

<sup>1</sup>Faculty of Information Science and Arts, Osaka Electro-Communication University, 1130-70 Kiyotaki, Shijonawate, 575-0063, Japan

<sup>2</sup>Graduate School of Information Science and Arts, Osaka Electro-Communication University, 1130-70 Kiyotaki, Shijonawate, 575-0063, Japan

Email: [nakahara@osakac.ac.jp](mailto:nakahara@osakac.ac.jp), [ht24a001@oecu.jp](mailto:ht24a001@oecu.jp)

\*Corresponding Author

## ARTICLE INFO

### Article History

Received 29 December 2025

Accepted 3 June 2026

### Keywords

Generative AI  
Artificial Language  
Code Generation  
UML

## ABSTRACT

In recent years, the shortage of IT professionals has become a critical issue globally, and particularly in Japan. While tools like GitHub Copilot improve efficiency for existing programmers, they often require a certain level of coding proficiency, leaving non-programmers or novice engineers unsupported. This study proposes a novel program generation method designed for novices with logical thinking skills but limited coding experience. The method utilizes a strictly defined "artificial language" (a constrained pseudo-flowchart) as a prompt for Generative AI, enabling users to generate code by focusing on logical flow rather than syntax. Empirical experiments compared the proposed method with manual coding by beginners. The results demonstrated that the proposed method achieved a 97% success rate compared to 41% for manual coding and significantly reduced average development time. These findings suggest that the proposed methodology can effectively convert non-programmers into active contributors in the development process.

© 2026 The Author(s). Published by The Society of Artificial Life and Robotics.

This is an open access article distributed under the CC BY-NC 4.0 license

(<http://creativecommons.org/licenses/by-nc/4.0/>).

## 1. Introduction

In recent years, the shortage of skilled labor has become a severe challenge globally. This issue is particularly acute in the IT industry in Japan. While demand is rising across all fields, the population of younger individuals receiving education to become IT professionals with the necessary knowledge continues to decline due to the aging population and low birthrate. For instance, in Japan, the shortage of personnel has become a more serious problem than ever before. According to projections by the Ministry of Economy, Trade and Industry, a shortage of approximately 790,000 workers is expected by 2030 [1]. Against this backdrop, the Ministry of Health, Labor and Welfare is considering countermeasures to address the shortage of IT personnel. These include securing side-job workers through the introduction of remote work, which is highly compatible with the IT industry, and utilizing ICT (*Information and Communication Technology*) and AI (*Artificial Intelligence*) for tasks such as automating simple operations through robots and automation [2]. Among these, generative AI capable of automating even complex tasks is drawing significant attention. It is expected to be utilized for enhancing efficiency and reducing labor in system development, where affinity with

automation and demand are high [3]. To achieve these efficiency gains and labor savings, the open-source development platform GitHub employs OpenAI's generative AI. This enables features such as suggesting the next code to write via autocomplete during source code creation and automatically generating portions of code [4]. However, there are also system engineers who cannot write programs but can create UML (*Unified Modeling Language*) documents such as specifications and flowcharts, as well as individuals who, despite having acquired the knowledge necessary for program creation, were forced to pursue careers outside the IT industry due to a lack of confidence in their abilities. This study focuses not on all individuals working outside the IT industry, but specifically on those who can understand business requirements and logically decompose processing steps, and who can diagram business processes using UML activity diagrams or flowcharts, yet struggle with the implementation phase involving syntax description and debugging in specific programming languages (hereinafter referred to as non-programmers).

This study does not aim to quantify the scale of non-programmers or their contribution to the talent shortage. Instead, it seeks to present technical measures for connecting the logical design capabilities possessed by

non-programmers to development outcomes, demonstrating the potential to lower barriers to participation in the development process. While such barrier reduction could ultimately contribute to alleviating the talent shortage, quantifying this contribution remains a future task.

Existing research has developed generative AI-based programming support technologies to enable such individuals to thrive in the IT industry [5][6], and investigated how introducing generative AI affects programmers' productivity [7]. The former research automatically generates programming hints and solutions for errors occurring within programs using generative AI. These hints have improved accuracy compared to conventional methods by learning from both error-prone programs and corrected programs. The latter study reported that using a specially developed large language model (LLM) showed little impact on productivity for experienced developers, but did improve productivity for junior developers. This suggests that the effectiveness of using generative AI to support IT professionals has been demonstrated, as experienced developers rely heavily on coding based on their own experience and tend to use LLM-based tools less frequently.

These existing studies require users to write a certain amount of program source code as a prerequisite, making it difficult to support system engineers who cannot write large amounts of source code or individuals who feel uncomfortable doing so. Therefore, this study proposes a program generation method using generative AI that assumes the creation of flowcharts, which impose less cognitive burden than directly writing program source code under specific conditions. Writing source code requires not only an understanding of algorithms but also knowledge of the specific, strict syntax, reserved words, and library functions of the programming language. For beginners and those out of practice, minor syntax errors like missing semicolons or mismatched parentheses typically result in compilation failures, significantly hindering development. On the other hand, flowcharts represent the logical flow and structure of a process using visual diagrams. This frees users from the need for strict syntax knowledge, allowing them to focus on the problem-solving logic itself. In other words, flowcharts specialize in defining a program's "what to do," thereby removing the language-specific barrier of "how to write it." This separation of concerns is precisely why creating flowcharts presents a lower hurdle for those who find reading and writing source code difficult. This approach is expected to enable personnel who primarily engaged in creating software design documents, or those who learned programming in the past but feel uncomfortable with it, to support development teams. The term "non-programmers" as defined here refers to personnel possessing the ability to understand business requirements, logically decompose processing steps, and diagram business processes using UML activity diagrams or flowcharts. While they possess experience defining system specifications and creating design documents, they struggle with memorizing and

writing the strict syntax required by specific programming languages like C++ and Python, deciphering and correcting error messages returned by compilers, and selecting and utilizing the appropriate API from vast libraries to achieve their objectives. Specifically, the individuals targeted for support in this research are those who possess logical thinking skills but face challenges in converting that thinking into code through implementation. Therefore, the objective of this study is to propose a method for narrowing the gap between the "logical design" of non-programmers and the "implementation results" demanded by development environments. This is achieved by representing the logical construction abilities possessed by non-programmers as flowcharts and converting their content into code using generative AI. This research does not propose a new algorithm for generative AI. Its novelty and practical value lie in designing an input format that is easy for non-programmers to handle and an interactive development workflow, based on existing generative AI.

This paper first investigates existing research related to source code writing support in Section 2, organizing the challenges encountered when applying such research to support non-programmers and the proposed countermeasures for these challenges. Next, Section 3 proposes a methodology for program creation support using artificial languages and generative AI tailored for non-programmers. Section 4 confirms the usefulness of the proposed methodology through empirical experiments. Finally, Section 5 presents a summary of the research and future prospects.

## **2. Classification of Programming Support Technologies and Literature**

### **2.1. Classification of Programming Support Technologies**

This section reviews existing research related to supporting program development. It then organizes the challenges that arise when applying these approaches to support development by non-professionals, along with strategies for addressing those challenges.

### **2.2. Literature Survey**

Existing research has proposed methods such as advising on how to correct errors during debugging [6], constructing programs interactively using natural language [8][9], providing thought processes or specific procedures in advance [10][11], and learning flowcharts converted into unique strings [12].

A technique for advising on how to fix errors during debugging [6] enables GPT-4o to generate explanations of bug causes and fixes from program source code by training it on detailed program descriptions and the code before and after corrections. Reports indicate this method provides feedback with accuracy comparable to human analysis. However, as it is designed to assist during debugging, it cannot be applied when developing new programs.

Interactive programming techniques using natural language [8][9] enable the construction of programs through exchanges with chatbot-style generative AI via natural language. Using these techniques, even beginners can automatically generate source code for simple programs. However, when developing programs with complex structures, it is difficult to automatically generate all the source code with a single prompt [13]. Therefore, it is necessary to repeatedly issue prompts and make corrections. However, as complexity increases, users may struggle to determine what prompts to input. Furthermore, during repeated instructions, source code that does not require editing may be altered for the worse, potentially preventing the successful construction of the intended program.

As a method for providing the thought process and specific steps in advance, a technique based on CoT (Chain-of-Thought) [10] has been proposed. These methods enable obtaining results closer to the user's desired form by providing examples and constraints to the LLM when constructing programs via natural language prompts, thereby facilitating algorithmic understanding. However, relying solely on the CoT approach is unsuitable for constructing complex programs, as the number of examples and constraints required to be provided to the LLM beforehand becomes enormous. Therefore, a method utilizing Structured Chain-of-Thought (SCoT), an evolution of CoT, has also been proposed [11]. SCoT is a method that uses LLMs to generate programs by inputting prompts in natural language describing steps more detailed than CoT, such as the types of input/output parameters and iterative processes, within the source code. The proposed SCoT method has been reported to generate programs with higher accuracy than CoT. Therefore, it can be said that inputting the detailed structure of the program can improve the accuracy of program generation. However, since prompts must be written into the source code, understanding the automatically generated source code is a prerequisite.

A method for learning flowcharts converted into unique strings [12] processes flowchart images using OCR (Optical Character Recognition) and shape recognition technology to convert the processing content into unique strings. It then automatically generates Python source code by pre-learning these converted strings. Using this method allows automatic generation of source code if it matches the scale or similar processing content of the pre-learned source code. However, to handle the generation of complex and lengthy source code, it requires training on an enormous number of flowchart patterns.

### **2.3. Identification of Key Challenges in Existing Research**

The findings indicate that using existing methods, it is highly likely that complex programs can be automatically constructed via natural language prompts when dealing with simple source code or code that is

reasonably decipherable. However, when supporting development by non-professionals, three challenges exist.

The first issue is the challenge of constructing prompts that require knowledge of source code. This problem arises in existing generative AI support tools when users cannot understand the generated source code and create accurate correction instructions as natural language prompts, thereby hindering the interactive development process. Existing approaches [6][10][11] assume users can read and interpret pre-developed or auto-generated source code to fix errors and iterate on fixes. However, non-experts often struggle to decipher source code, making these assumptions a significant barrier to effective use. Therefore, an input method that does not require source code knowledge for prompt creation is necessary.

The second issue is the “challenge of constructing complex prompts.” This problem fundamentally refers to the difficulty users face in consistently devising and writing appropriate, specific prompts to fulfill their own requirements. Existing research [8][9] assumes that prompts input to generative AI are provided by developers. However, this can lead to outputs differing from the intended purpose, or even similar prompts producing entirely different outputs. Furthermore, users may simply be unable to conceive of an appropriate prompt in the first place. Therefore, a method is needed that can consistently produce acceptable results without relying on the accuracy of prompt creation.

The third point is the “challenge of generating complex source code.” This challenge refers to the problem that existing research methods are limited to generating small-scale code of a few lines, and cannot handle the generation of entire hierarchical and complex programs spanning tens to hundreds of lines required for practical development. Existing research [12] assumes prior learning of source code. However, creating and training on flowcharts for the diverse and vast amount of source code expected to be created in development environments is impractical. Furthermore, while existing research is limited to generating source code of a few lines, the lines of source code required in development environments range from tens to hundreds or more, making it difficult to represent them using a single flowchart. Therefore, it is necessary to utilize large language models (LLMs) like GPT-4o, trained on vast amounts of web-based source code, to enable the generation of complex programs composed of multiple flowcharts.

## **3. Proposal of a Coding Support Method Using Artificial Languages and Generative AI**

This section proposes a methodology to support the creation of programs that take into account the circumstances faced by non-programmers, based on the three challenges discussed in Section 2.

### 3.1. Approach to Addressing Issues with Existing Methods

For the first issue—“the challenge of constructing prompts requiring source code knowledge”—we address it by defining and inputting artificial languages that are interpretable by generative AI. For the second issue, “the challenge of constructing complex prompts,” we address it by providing methods for interpreting the artificial language and preparing template prompts. For the third issue, “the difficulty of generating complex source code,” we address it by improving the flowchart notation to enable hierarchical representation of both high-level and detailed flowcharts.

### 3.2. Overview of the Proposed Method

The processing flow of the proposed method is shown in Figure 1. The proposed method consists of two steps: creating a pseudo-flowchart and automatically generating the program. The input data is the requirements for the program to be developed, and the output data is the automatically generated source code. The requirements used in this study are the information necessary for constructing the program, such as the programming language, operational conditions, and calculation formulas used for computations.

### 3.3. Creating a Pseudo Flowchart

In this step, developers manually create pseudo-flowcharts based on the program requirements. To unambiguously define the elements constituting the artificial language proposed in this study—namely, the pseudo-flowchart—and their interpretation, notation rules are established.

Figure 2 illustrates the six types of symbols defined in this language. Circles indicate the start and end points of the entire flowchart process. Rectangles describe specific processing actions, such as assignments or calculations, in natural language. Parallelograms represent input/output operations, like keyboard input or screen output. Diamonds describe decision conditions that branch the flow of processing. Finally, trapezoids with a short top edge and their inverse define the start and end of a loop, respectively, indicating the scope of a series of repetitive processing steps.

Strict rules shall be established for decision symbols and their associated flows. The conditional expressions written within decision symbols must yield a result that is uniquely determined as either true (Yes) or false (No). Furthermore, two output arrows must always be drawn from decision symbols, with the flow for the true condition indicated by a red arrow and the flow for the false condition indicated by a blue arrow. This prevents generative AI from misinterpreting branching intent. Due to this rule, this language does not permit the direct representation of multi-branching with three or more branches; instead, multiple decision symbols are combined as necessary to express it.

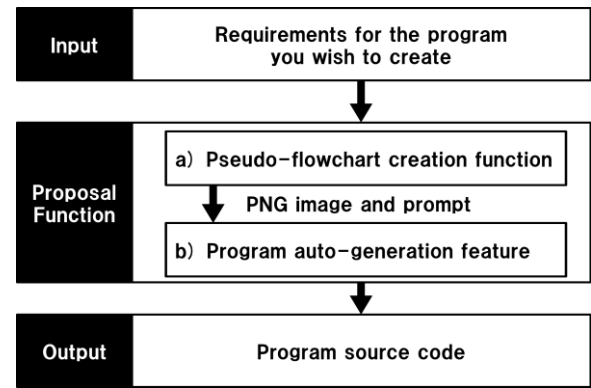


Figure 1 Proposed Method Flow

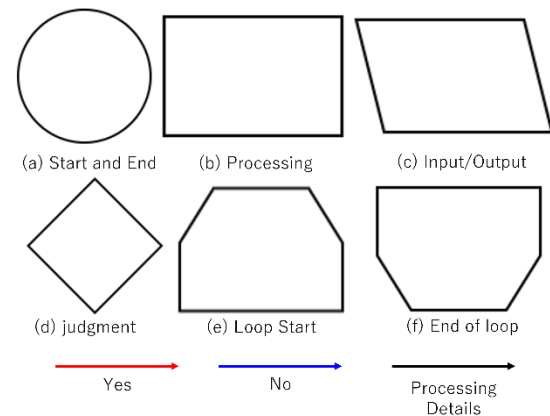


Figure 2 Examples of Shapes used in Pseudo-Flowchart

The hierarchical structure, a major feature of this language, also clearly defines its relationships. The relationship between the “Process” symbol in the outline flow and its corresponding “Detailed Flow” corresponds to subroutine or method calls in programming. This means replacing one process in the schematic flow with a more detailed sequence of processing flows. This connection is represented by extending a black arrow from any “Process” symbol in the schematic flow and connecting it to the starting point of the corresponding detailed flow. Furthermore, since this language permits nested hierarchies, it is possible to further detail a process within a detailed flow by using another detailed flow. Similarly, hierarchical nesting of process blocks within loop processing is also permitted. The hierarchical structure described in this manner is interpreted as a single program as a whole and is reflected in the structure generated by the AI when producing source code (e.g., private methods or code blocks within functions).

When describing conditional branching in traditional UML, “yes” or “no” is written above the arrow. However, when inputting this notation into a flowchart generation AI, it tended to ignore the ‘yes’ or ‘no’ text and output source code that branches under the opposite condition. This tendency was observed regardless of whether the input format to the generation AI was XML or image format. Existing research [14][15] exploring the potential of deep learning and machine learning for automatic diagram

interpretation also pointed out challenges in associating shapes with text strings. Therefore, we concluded that the tendency observed in preliminary experiments stemmed from an inability to understand the spatial relationship and connection between arrows and nearby text. To prevent misinterpretation, we changed the notation to one where the color of the arrow indicates affirmative or negative conditions.

When creating flowcharts for complex programs, it is common to first describe the overall processing flow and then detail the flow for each individual process. However, dividing the processing flow into multiple parts often leads to complex prompts required to connect them, making it highly likely that the intended program will not be generated. Therefore, the pseudo-flowchart proposed in this study improves the notation, as shown in Figure 3, to connect the general processing flow with the detailed flows of each process. This allows the details to be described hierarchically within a single flowchart. If the subsequent automatic program generation results differ from expectations, the program can be refined by improving this processing flow and repeating the regeneration process.

Based on the above, this step enables interaction with generative AI through pseudo-flowcharts rather than source code, thereby offering potential improvements to the challenge of constructing prompts requiring source code knowledge. Furthermore, by enabling hierarchical representation of both high-level and detailed flowcharts within a single flowchart, it offers potential improvements to the challenge of generating complex source code.

### 3.4. Automatic Program Generation

In this step, we input the created pseudo flowchart into the generative AI to automatically generate the program. The prompt for inputting into the generative AI is shown in Figure 4. At this time, the prompt should include the programming language for which we want automatic generation, the language used for comments and display, and the meaning for each arrow color. By inputting both this prompt and the pseudo flowchart simultaneously, the source code is automatically generated. Consequently, once the pseudo flowchart is created, the prompt becomes a standardized template. This eliminates the need for developers to modify the prompt content based on the complexity of the program structure. Therefore, this step is expected to address the challenge of “inability to construct complex prompts.”

### 3.5. The Suitability of This Language's Expressiveness for Its Purpose

The pseudo-flowchart proposed in this study may appear to have more constraints than standard flowchart notation at first glance. However, it possesses sufficient expressive power for the purposes of this research, and these constraints are based on intentional design. First, we describe the fundamental expressive power of this language. The Structured Theorem demonstrates that any computable algorithm can be expressed as a combination

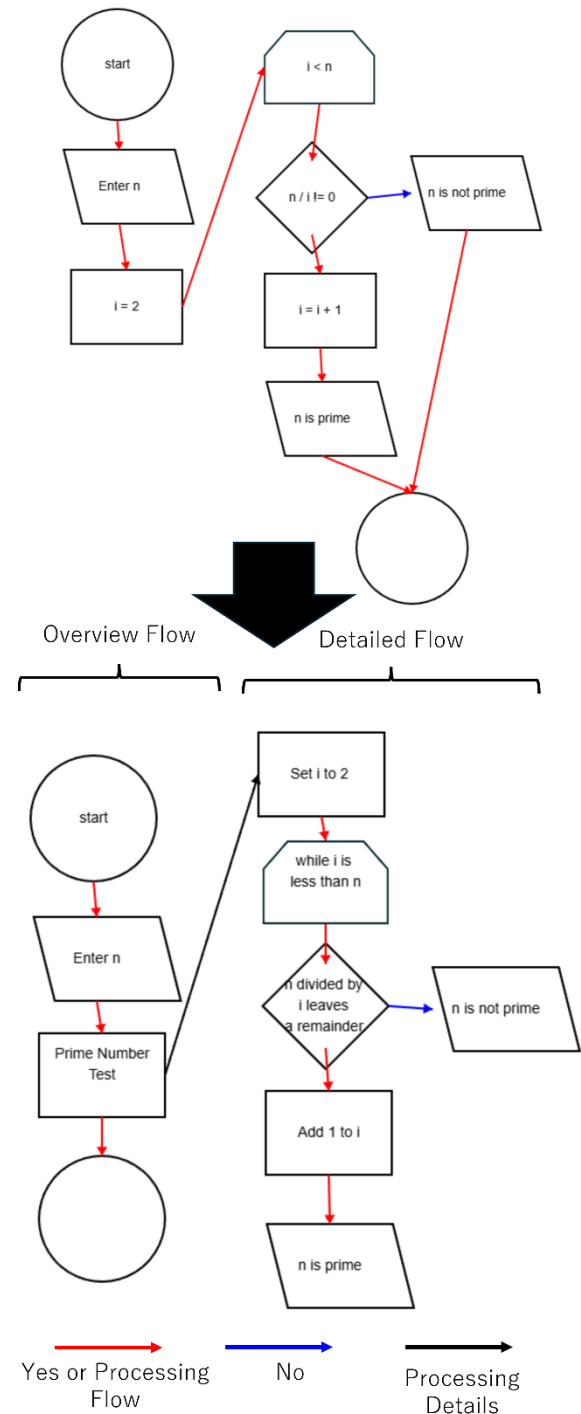


Figure 3 Separate Description Method for Schematic and Detailed Flows

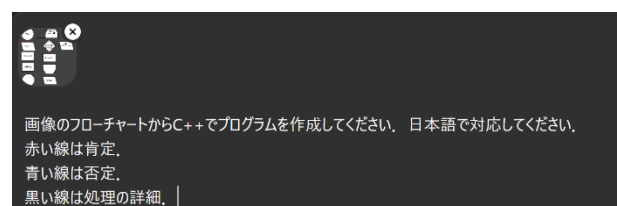


Figure 4 Example of Prompts to be Inputted into a Generative AI (Japanese Prompt was Used)

of three basic control structures: “sequential,” “branch,” and “loop.” This language can explicitly express sequential structures through concatenation of “process” symbols, branching (if-then-else) structures using ‘decision’ symbols limited to two values, and iterative (while, for) structures using “loop start/end” symbols. Because it can describe these three fundamental structures, this language possesses the fundamental capability to express any algorithm.

Next, we explain the constraint that this language does not permit multi-branching or free connections (equivalent to so-called goto statements). This constraint is an essential design element for achieving the goal of this research: “accurately converting the logical structure described by non-programmers into code using generative AI.” For instance, limiting branching to binary decisions and fixing meaning through color serves to thoroughly eliminate ambiguities that could confuse the generative AI’s interpretation. Furthermore, prohibiting free connections and enforcing structured descriptions prevents the emergence of complex, hard-to-read flowcharts resembling spaghetti code. This is also expected to enhance the quality and maintainability of the resulting generated code.

Based on the above, the expressive power of this language is not aimed at describing any program without restriction. Instead, it is specifically designed to express the logical thinking possessed by the untrained individuals targeted by this research in a structured and unambiguous manner. This intentional constraint achieves an optimal balance between expressive freedom and the accuracy of AI interpretation, providing sufficient and appropriate expressive power to achieve the objectives of the proposed method.

Furthermore, frameworks such as Model-Driven Development (MDD) exist for generating code from models. MDD relies on formalized model descriptions and toolchains, excelling in generating code based on the rigor and consistency of the model. In contrast, the untrained personnel targeted in this research are assumed to possess a relative advantage in the ability to decompose business requirements into procedures and represent them schematically, rather than in acquiring and applying formal specifications or tool-specific notations. This research aligns with this characteristic by using flowcharts—a familiar diagrammatic representation—as input. It aims to connect to implementation outcomes while minimizing learning and operational costs by having generative AI handle the mapping to the implementation language. Therefore, this research is not intended to replace MDD but is positioned as a complementary approach with different target personnel, prerequisite skills, and ease of introduction.

### 3.6. The Suitability of This Language's Expressiveness for Its Purpose

This research proposes not a unidirectional code generation technique, but a development support method

where users and generative AI interactively complete programs together. The anticipated workflow for users constructing programs is shown below.

#### Step 1: Creating a Pseudo Flowchart

Users first create a pseudo-flowchart based on the requirements of the program they wish to develop, following the notation rules defined in this research. This is the process of expressing the user's thoughts and design intent in this language.

#### Step 2: Generating Source Code

The user inputs the image of the created pseudo-flowchart and the template prompt shown in [Figure 4](#) into the generative AI to generate the initial version of the source code.

#### Step 3: Verifying Functionality

The user compiles and executes the generated source code to confirm whether it functions as required or whether any compilation errors occur.

#### Step 4: Analysis of Results and Feedback

If the program does not behave as expected during functional verification, the user analyzes the cause and provides feedback. It is crucial that the user does not directly edit the generated source code at all. Feedback must always be provided by modifying the input pseudo-flowchart. The following are the anticipated scenarios:

- Case A: Error in User Intent (Flowchart)  
When the generated code follows the flowchart's description, but the flowchart's logic itself is incorrect (e.g., incorrect loop termination condition, reversed branch condition). In this case, the user corrects the logical structure of the pseudo-flowchart.
- Case B: Misinterpretation by Generative AI  
The logic of the flowchart is correct, but if the generative AI misinterprets its intent (e.g., incorrect ESC key assignment as described later in Section 4.3), the user should revise the flowchart description to be clearer and more specific to prevent misunderstanding by the AI. Examples include rewriting “until the ESC key is pressed” as “until key code 0x1B is input,” or splitting one complex process into multiple simpler ones.

#### Step 5: Improvement through Regeneration

The user returns to Step 2 using the modified pseudo-flowchart and regenerates the source code. This cycle of “Generate → Verify → Modify Flowchart → Regenerate” is repeated until the program fully satisfies the requirements.

Thus, this method does not force users to debug source code directly, but instead encourages trial and error on a pseudo-flowchart that serves as a blueprint of their own thinking. This enables even users unfamiliar with programming syntax to interactively complete programs at the level of logical design, where they excel.

Table 1 Requirements for Programs Used for Evaluation

| Problem Number | Program Requirements (Overview)                                                                                                 | Estimated Lines of Code |
|----------------|---------------------------------------------------------------------------------------------------------------------------------|-------------------------|
| 1              | Convert the entered temperature to Celsius (°C), Fahrenheit (°F), or Kelvin (K)                                                 | 40                      |
| 2              | Encrypt the entered content using Caesar cipher                                                                                 | 40                      |
| 3              | Generate a Fibonacci sequence starting from the entered number                                                                  | 50                      |
| 4              | Calculate the entered arithmetic operations or simultaneous equations                                                           | 80                      |
| 5              | Search for prime numbers within the entered set of numbers. If no prime is found, display the closest prime                     | 90                      |
| 6              | Calculate the price of a dish based on inputted ingredient costs, usage fees, and fixed costs to achieve a target profit margin | 100                     |
| 7              | Calculate the percentage of expenses based on inputted monthly income and expenses                                              | 120                     |
| 8              | Determine pass/fail status for each subject based on inputted student names and subject-specific grades                         | 150                     |
| 9              | Display how many days until stock runs out based on inputted product inventory levels and consumption rates                     | 170                     |
| 10             | Manage the lending status of each book by inputting book data                                                                   | 200                     |

## 4. Verification Experiment

### 4.1. Experimental Details

In this experiment, to verify the usefulness of the proposed method, we compared whether 10 beginners could construct programs with and without using the proposed method, the success of the generated source code, and the time required for construction. The requirements for the programs used in the evaluation are shown in Table 1, which details the programming language, operational conditions, and specific calculation formulas for each task. Ten problems were prepared for the evaluation, designed such that the complexity and amount of code required for implementation increased as the problem number progressed.

When constructing programs using the proposed method, subjects first create pseudo-flowcharts aligned with each program's requirements. These pseudo-flowcharts are then input into the generative AI. Subjects subsequently verify whether the output program functions correctly. If errors occur, subjects edit the pseudo-flowchart and repeat the regeneration process until the program operates correctly. At this stage, subjects were not expected to decipher the generated source code. Conversely, when subjects develop programs without using the proposed method, they write the source code independently. While internet searches are permitted, natural language-based generative AI assistance is not used. This approach allows for clear observation of how much the proposed method improves task continuity and completion likelihood for the specific situation addressed in this study: “the inability to progress with source code writing itself.”

However, in practical implementation, the option of using generative AI in natural language for implementation support is also realistic. Therefore, this paper does not address comparisons with “conditions for freely using generative AI in natural language for implementation,” which remains a future task.

In this experiment, requirements were presented to subjects electronically in the format shown in Figure 5 for each problem. Descriptions were kept objective to avoid

### Temperature conversion

- Programming language: C++
- Convert between Celsius (°C), Fahrenheit (°F), and Kelvin (K)
- Operates via keyboard input
- Allow selection of the temperature to be converted
- Display the resulting temperature
- Continue until the “Esc” key is pressed
- Fahrenheit = Celsius  $\times 9/5 + 32$
- Kelvin = Celsius + 273.15

Figure 5 Examples of Requirements Presented in the Experiment

advantages or disadvantages based on the method. Success was defined as being able to execute the generated or written program and confirm it performed as per the presented requirements. Failure was defined as the subject determining further creation was impossible or the program executing with behavior differing from the presented requirements. To prevent premature giving up, subjects were instructed to attempt the task for at least 10 minutes. The “construction time” measured in this experiment refers to the total time from when the subject was presented with the problem requirements until they completed a functioning program that satisfied those requirements. This includes all time spent reading the problem, considering algorithms, and creating and modifying (debugging) the deliverable (source code or pseudo-flowchart).

In this experiment, both the proposed method and manual program creation were tasked with solving the same 10 problems. To prevent bias from a specific method consistently being assigned later, half of the participants (5 people) created odd-numbered problems using the proposed method and even-numbered problems manually. The remaining 5 participants created odd-numbered problems manually and even-numbered problems using the proposed method.

### 4.2. Experimental Conditions

The subjects for this experiment were 10 individuals who had not previously worked in the field. Subjects

Table 2 Success or Failure of Program Build

| Build Results       | Proposal Method | Manual  |
|---------------------|-----------------|---------|
| Success             | 97              | 41      |
| Failure             | 3               | 59      |
| Longest Build Time  | 1:20:00         | 2:45:00 |
| Shortest Build Time | 0:02:00         | 0:05:00 |
| Average Build Time  | 0:19:00         | 0:48:00 |
| Success Rate        | 97%             | 41%     |

Table 3 Build Time per Problem using Proposed Method

| Problem Number | Maximum Build Time | Minimum Build Time | Average Build Time | Average Number of Build Runs |
|----------------|--------------------|--------------------|--------------------|------------------------------|
| 1              | 1:20:00            | 0:04:00            | 0:27:00            | 61                           |
| 2              | 0:45:00            | 0:02:00            | 0:14:00            | 42                           |
| 3              | 0:40:00            | 0:03:00            | 0:14:00            | 37                           |
| 4              | 1:11:00            | 0:03:00            | 0:25:00            | 81                           |
| 5              | 0:47:00            | 0:02:00            | 0:18:00            | 69                           |
| 6              | 1:01:00            | 0:03:00            | 0:22:00            | 99                           |
| 7              | 1:13:00            | 0:02:00            | 0:20:00            | 96                           |
| 8              | 1:15:00            | 0:02:00            | 0:16:00            | 85                           |
| 9              | 0:51:00            | 0:02:00            | 0:12:00            | 78                           |
| 10             | 1:00:00            | 0:06:00            | 0:25:00            | 146                          |

possessed knowledge of C++ gained through coursework or similar learning experiences, understanding basic grammar concepts such as conditional branching and loops, but were at a level where writing source code from scratch independently was difficult. All subjects received a lecture on how to write pseudo-flowcharts prior to participating in this experiment. Regardless of the presence of the proposed method, the development language was C++. Prompts to the AI were input in Japanese. Images input to the generative AI were PNG files to minimize image degradation. The generative AI used was ChatGPT, which was investigated in Section 2.2 among existing studies, confirmed to be useful for program construction support, and employed here. The ChatGPT version used was GPT-4o, which supports image input and demonstrates high accuracy in typical tasks. The order of the proposed method and manual development was alternated for each problem number. Furthermore, the order was reversed for each participant to minimize the influence of the proposed method or manual development.

### 4.3. Results and Discussion

Table 2 summarizes the total number of successful and unsuccessful program constructions for all subjects using both the proposed method and the manual method. As shown in Table 2, the manual method achieved a success rate of only 41%, whereas the proposed method achieved a significantly higher success rate of 97%. Furthermore, comparing the time required for construction, the proposed method completed coding in less time for the longest, shortest, and average durations. This difference in success rates stems from the cognitive load of the task under both methods and the differing responses of subjects upon

failure. In this experiment, “failure” was defined as “being unable to create source code.” Most failures in manual development occurred when subjects became stuck during coding and ultimately gave up. Subjects with less than one year of development experience, in particular, failed to complete a single program using manual development (Table 6). This is likely because the task of writing code from scratch imposed excessive cognitive load—such as not knowing how to implement an algorithm or encountering unsolvable syntax errors—preventing them from continuing their attempts. In contrast, the failure rate using the proposed method was limited 3%. Participants first began by concretizing their thoughts as pseudo-flowcharts.

Even when the generated program did not function as intended, they did not directly modify the source code. Instead, they returned to the pseudo-flowchart, the foundation of their thinking, revised its logical structure, and attempted regeneration. This “flowchart-based trial and error” was less taxing than wrestling with source code errors, allowing participants to make multiple attempts without giving up. As a result, the average construction time was significantly reduced to 19 minutes, compared to 48 minutes manually. From the above analysis, the proposed method not only shortens coding time but also reduces cognitive load during development, enabling repeated trials to overcome failures. This makes it a powerful support tool for program construction, especially for beginners and non-programmers. Consequently, it is considered to have also improved the “challenge of constructing complex prompts.”

The time required to construct the program for each problem is shown in Table 3 and Table 4. Comparing the

Table 4 Build Time per Problem by Manual

| Problem Number | Maximum Build Time | Minimum Build Time | Average Build Time | Average Number of Build Runs |
|----------------|--------------------|--------------------|--------------------|------------------------------|
| 1              | 0:40:00            | 0:13:00            | 0:23:00            | 46.40                        |
| 2              | 0:25:00            | 0:5:00             | 0:16:00            | 30.80                        |
| 3              | 0:21:00            | 0:8:00             | 0:12:00            | 25.60                        |
| 4              | 2:00:00            | 0:52:00            | 1:26:00            | 151.33                       |
| 5              | 0:50:00            | 0:15:00            | 0:31:00            | 66.80                        |
| 6              | 2:15:00            | 0:22:00            | 1:06:00            | 124.00                       |
| 7              | 2:14:00            | 0:16:00            | 1:20:00            | 178.25                       |
| 8              | 1:05:00            | 0:31:00            | 0:43:00            | 78.00                        |
| 9              | 1:24:00            | 0:12:00            | 0:42:00            | 96.50                        |
| 10             | 2:45:00            | 0:15:00            | 1:20:00            | 159.67                       |

Table 5 Results for One-Year's Development Experience

| Problem Number | Proposal Method | Manual     |
|----------------|-----------------|------------|
| 1              | 0:51:00         | Unfinished |
| 2              | 0:45:00         | Unfinished |
| 3              | 0:40:00         | Unfinished |
| 4              | 1:11:00         | Unfinished |
| 5              | 0:46:00         | Unfinished |
| 6              | 1:01:00         | Unfinished |
| 7              | 1:13:00         | Unfinished |
| 8              | 1:15:00         | Unfinished |
| 9              | 0:51:00         | Unfinished |
| 10             | 1:00:00         | Unfinished |

Table 6 Results for Two-Years' Development Experience

| Problem Number | Proposal Method | Manual  |
|----------------|-----------------|---------|
| 1              | 0:13:00         | 0:13:00 |
| 2              | 0:09:00         | 0:05:00 |
| 3              | 0:06:00         | 0:08:00 |
| 4              | 0:18:00         | 0:52:00 |
| 5              | 0:13:00         | 0:32:00 |
| 6              | 0:22:00         | 1:24:00 |
| 7              | 0:11:00         | 2:15:00 |
| 8              | 0:11:00         | 0:31:00 |
| 9              | 0:07:00         | 0:59:00 |
| 10             | 0:12:00         | 1:34:00 |

longest construction times, for Problems 1 to 3, manual construction was shorter than the proposed method, while for Problem 4 and beyond, the times were comparable or manual construction tended to take longer than the proposed method. Regarding the shortest construction time and average time, as the problem number increased—meaning the amount of code to be written increased—the time required for development also tended to increase. This is because when the amount of code required for implementation is small, manually creating the source code is faster. However, as the amount of required code increases, developers must proceed by verifying their own written code, leading to increased development time. Based on these findings, the proposed method is expected to contribute to improving the efficiency of constructing programs with complex structures.

Table 3 shows that we were able to automatically generate source code ranging from 37 to 146 lines. While existing research [11] reports generating only a few lines of Python code, the line count varies across programming languages even for identical functionality, making a simple comparison based solely on line count inappropriate. What is crucial for evaluating the usefulness of this research is not the number of lines of code, but rather the “program completion rate by non-programmers,” which is the objective of this study. Indeed, as shown in Table 5, participants with less than one year of development experience could not complete a single program manually. However, using this method, they were able to complete programs with practical functionality—such as student grade management (Problem 8) and inventory management (Problem 9)—that they likely would have

found difficult to even begin implementing on their own. This demonstrates that, unlike support for “developers who can handle source code” targeted by many existing studies, this method functions as a means to bridge the critical gap between “non-programmers who cannot handle source code” and the development environment. Herein lies the novelty and practical value of this research. Therefore, it can be said that the “challenge of generating complex source code” has been improved.

The time required to construct programs by years of development experience is shown in Table 5 and Table 6. “No record” in the tables indicates problems where the subject could not construct the program. For the 2-year group, as mentioned earlier, development time tends to increase with problem number when using manual methods. In contrast, subjects with less than 1 year of development experience could not write source code for any problem using manual methods. However, using the proposed method, they were able to construct all programs, albeit taking longer than the two-year experience group. Furthermore, examining the source code generated by the proposed method revealed similar structures across all subjects. This suggests that using pseudo-flowcharts and specified prompts can ensure the source code generated by ChatGPT maintains a consistent level of quality. These results indicate that even individuals without prior experience or knowledge of source code can potentially support development environments. Therefore, we can conclude that the “challenge of constructing prompts requiring source code knowledge” has been addressed.

Upon examining the program generated by the proposed method, we observed that even when upper and lower bounds for numerical values were not explicitly specified in the pseudo-flowchart, error handling for input values outside the specified range was automatically added to the generated program. This is likely because the source code GPT-4o was pre-trained on the web often included safety-conscious error handling, prompting it to automatically add such handling. This result suggests the proposed method generates source code patterns similar to those found in analogous web source code, based on the flowchart's content.

Upon reviewing the programs created by all subjects using the proposed method, no issues were found with either input or output. However, as shown in Figure 6, despite specifying in the pseudo-flowchart that pressing the ESC key should trigger an action, instances were observed where a different key was assigned instead. Furthermore, it was confirmed that each time the same flowchart was input into ChatGPT, a different button was assigned. This suggests that GPT-4o cannot accurately recognize the characters and their meanings within the pseudo-flowchart. Instead, it statistically processes the publicly available source code used as training data, randomly assigning key input events that are frequently specified. Furthermore, it is highly likely that GPT-4o fails to understand that the ESC key and the code 0x1B (assigned to the ESC key in the source code) represent the same meaning. To address this issue, improvements can be

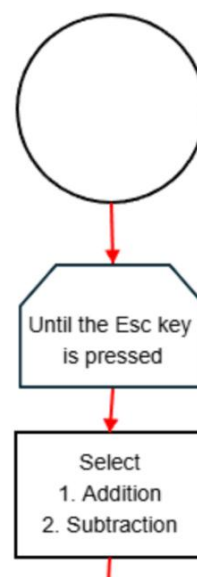


Figure 6 Part of a Pseudo-Flowchart Created by the Subject

expected by exploring methods to describe key input events using machine-readable symbols, similar to the machine-readable arrow colors proposed in this study.

## 5. Conclusion

This study proposes a development support method that generates source code using generative AI, taking pseudo-flowcharts describable by non-programmers as input and employing standardized prompts. Through experiments, we confirmed that using the proposed method tends to improve completion probability and task persistence compared to manual coding for individuals who find writing source code from scratch difficult. This resolves the challenges of existing methods—namely, the need for source code knowledge to construct prompts and the inability to construct complex prompts—thus validating the usefulness of our approach. Furthermore, while much existing research focuses on enhancing the productivity of experienced developers, by shifting the focus of interaction from source code to flowcharts, we have proposed a novel development method that demonstrates a pathway to incorporate individuals who previously found it difficult to participate in the development process as new contributors.

Going forward, we plan to refine the definition of pseudo-flowcharts to address key input events, which GPT-4o struggles with as revealed in this study. Furthermore, this research focused solely on console programs. Therefore, we need to clarify the applicability of our approach to software development involving UI elements with more complex structures than console programs, as well as to web system development. In doing so, we plan to aim for more advanced automated generation by combining our proposed method with techniques using generative AI specialized for UI generation [16].

## References

1. Ministry of Economy, Trade and Industry, "Reference Material (Status of IT Human Resource Development, etc.)," [https://www.meti.go.jp/shingikai/economy/daiyoji\\_sangyo\\_skill/pdf/001\\_s02\\_00.pdf](https://www.meti.go.jp/shingikai/economy/daiyoji_sangyo_skill/pdf/001_s02_00.pdf) (in Japanese).
2. Ministry of Health, Labour and Welfare, "Annual Health, Labour and Welfare Report 2024, Section 2: Addressing Labor Shortages," <https://www.mhlw.go.jp/stf/wp/hakusyo/roudou/24/2-2.html> (in Japanese).
3. The Japan Research Institute, Limited, "Current Status and Prospects of System Development Using Generative AI - Towards System Development for the Generative AI Era -," <https://www.jri.co.jp/MediaLibrary/file/advanced/advanced-technology/pdf/15272.pdf>
4. "GitHub : GitHub Copilot", <https://github.com/features/copilot>
5. Zhu, Q., Luo, X., Liu, F., Gao, C. and Che, W. "A Survey on Natural Language Processing for Programming" LREC-COLING 2024, pp1690-1704, 2024.
6. Phung, T., Pădurean, V., Singh, A., Brooks, C., Cambronero, J., Gulwani, S., Singla, A. and Soares, G. "Automating Human Tutor-Style Programming Feedback: Leveraging GPT-4 Tutor Model for Hint Generation and GPT-3.5 Student Model for Hint Validation," Learning Analytics and Knowledge Conference, Vol.14, pp.12-23, 2024.
7. Gambacorta, L., Qiu, H., Shan, S. and Rees, M.D. "Generative AI and Labour Productivity: a Field Experiment on Coding, BIS Working Papers" No.1208, 2024.
8. Qasse, I., Mishra, S. and Hamdaqa, M. "Chat2Code: Towards Conversational Concrete Syntax for Model Specification and Code Generation, the Case of Smart Contracts", <https://arxiv.org/abs/2112.11101>
9. Pădurean, V., Denny, P., Gotovos, A. and Singla, A. "Prompt Programming: A Platform for Dialogue-based Computational Problem Solving with Generative AI Models", <https://arxiv.org/abs/2503.04267>
10. Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Le, Q. and Zhou, D. "Chain-of-thought Prompting Elicits Reasoning in Large Language Models", Proceedings of the 36th International Conference on Neural Information Processing Systems, Vol.36, No.1800, pp.24824-24837, 2022.
11. Li, J., Li, G., Li, Y. and Jin, Z. "Structured Chain-of-Thought Prompting Enhances Code Generation with Large Language Models", ACM Transactions on Software Engineering and Methodology, Vol.34, No.2, pp.1-23, 2025.
12. Shukla, S., Gatti, P., Kumar, Y., Yadav, V., Mishra, A. "Towards Making Flowchart Images Machine Interpretable", <https://arxiv.org/abs/2501.17441>
13. Tian, H., Lu, W., Li, O.T., Tang, X., Cheung, S., Klein, J. and Bissyandé, T.F. "Is ChatGPT the Ultimate Programming Assistant -- How far is it? ", <https://arxiv.org/abs/2304.11938>
14. Mani, S., Haddad, M.A., Constantini, D., Douhard, W., Li, Q. and Poirier, L. "Automatic Digitization of Engineering Diagrams Using Deep Learning and Graph Search" IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops, 2020.
15. Yang, L., Wang, J., Zhang, J., Li, H., Wang, K., Yang, C. and Shi, D. "Practical Single-Line Diagram Recognition Based on Digital Image Processing and Deep Learning", Expert Systems with Applications Vol.238, No.1, 2024.
16. Liu, Y., Sra, M. and Xiao, C. "CrowdGenUI: Aligning LLM-Based UI Generation with Crowdsourced User Preferences," <https://doi.org/10.48550/arXiv.2411.03477>

---

## Authors Introduction

Dr. Masaya Nakahara



He received the B.S. degree in informatics from Kansai University, Osaka, Japan, and the M.S. and Ph.D. degrees in knowledge information science from Kansai University, Osaka, Japan. From April 2021 to March 2025, he was a lecturer with the Faculty of Information Science and Arts at Osaka Electro-Communication University, Osaka, Japan. Since April 2025, he has been an associate professor with the same faculty. He has been awarded various research grants from Japan's NICT, and private companies. As a principal investigator or co-investigator, he has been engaged in research on digital twins, gamification, image processing and point cloud data.

Mr. Kaito Andachi



He received his bachelor's degree in informatics from Osaka Electro-Communication University in 2024 and is currently a graduate student at the same university.

---