Software: Practice and Experience

WILEY

RESEARCH ARTICLE OPEN ACCESS

NoCodeGPT: A No-Code Interface for Building Web Apps With Language Models

Mauricio Monteiro[1,2] | Bruno Castelo Branco[3] | Samuel Silvestre[3] | Guilherme Avelino[3] | Marco Tulio Valente[1]

[1]UFMG, Belo Horizonte, Minas Gerais, Brazil | [2]Department of Industrial Automation and Information Technology, IFMG, Betim, Minas Gerais, Brazil | [3]Department of Computer Science, UFPI, Teresina, Piauí, Brazil

Correspondence: Mauricio Monteiro (mauricio.monteiro@ifmg.edu.br)

Received: 29 February 2024 | **Revised:** 26 March 2025 | **Accepted:** 15 April 2025

Funding: This study was supported by grants from CNPq and FAPEMIG.

Keywords: automated software engineering | ChatGPT | large language models | NoCode tools | web development

ABSTRACT

Background: Language models are increasingly used by software developers. However, it remains unclear whether their standard chat-based interfaces are suitable for software development—especially for users with limited programming experience.
Objective: This work presents a tool, called NoCodeGPT, that provides a customized interface for language models aimed at enabling the implementation of small web applications without writing code.
Methods: We first conducted an exploratory study in which three participants used ChatGPT to implement a simple web-based application. After that, we designed and implemented a customized GPT interface, called NoCodeGPT. To evaluate this new interface, we asked 14 students with limited web development experience to build two small web applications using only prompts.
Results: The exploratory study showed that general-purpose chat interfaces like ChatGPT are not user-friendly for application development. One participant, for instance, was unable to complete any proposed user stories. In contrast, results with NoCodeGPT were encouraging: 9 out of 14 participants completed all user stories, while the remaining five completed at least half.
Conclusion: The standard GPT interface is not well-suited for novice web developers. In response, we proposed, designed, and implemented a new interface that offers a more accessible experience for building web applications with language models.

1 | Introduction

Large-scale Language Models (LLM) are experiencing significant adoption among software developers, with some studies reporting major improvements in productivity. For example, a recent study by GitHub, based on telemetry data from nearly one million users, concluded that developers tend to accept 30% of the code suggestions provided by the Copilot tool.[1] The study extrapolates that this adoption rate is equivalent to adding 15 million developers to the global workforce of software professionals. It also concludes that these productivity gains will have a significant impact as "developers seize new opportunities to utilize AI for solutions design and accelerate digital transformation worldwide."

Other studies investigated the usage and benefits of language models in specific software engineering tasks, including fixing bugs [1, 2], writing unit tests [3], writing code comments [4, 5], detecting code smells and technical debt issues [6, 7], and solving programming problems [8–10]. However, to the best of our knowledge, there are few papers that investigate the use of language models for the end-to-end construction of software systems, that is, in contexts where a developer has a set of requirements and has to design, implement, test, and validate an entirely new system [11]. In such contexts, language models are used as code generators, receiving as input prompts describing the functional and nonfunctional requirements of a system and producing as output a runnable software application. One exception is a paper by Peng et al. [12], in which the authors evaluated the use of GitHub Copilot to implement a web server based on a high-level textual description. On the one hand, they were among the first to measure, through a controlled study, the productivity gains achievable with AI tools. On the other hand, their target app was relatively small (a simple web server) and was written entirely in JavaScript. It did not follow widely adopted software architectures, such as those structured into front-end and back-end components, which are common in modern web-based systems. Furthermore, the implementation did not make use of popular frameworks for building web interfaces or managing databases. Another example refers to a demo conducted by one of the cofounders of OpenAI during the launch of GPT-4.[2] To demonstrate the power of the new version, he drew a mockup of an application on a napkin, and from a photo of that mockup, ChatGPT was able to generate a functional web application, although apparently consisting only of its front-end component.

In this article, we start by reporting an exploratory study using ChatGPT version 4 to implement a simple Web-based application. We defined a set of user stories and technologies for the system. Then, three developers with different profiles and experience independently attempted to implement this application using only ChatGPT.

As a key finding of this first study, it became clear that ChatGPT does not offer a user-friendly interface for building applications, even small web systems. In fact, the two participants with web development experience were able to build the proposed system. However, the third participant, with limited experience in software development, was unable to complete any of the proposed user stories. For example, he struggled to copy the code generated by ChatGPT into the correct folders (as defined by the system's architecture). More importantly, when ChatGPT produced an incorrect sequence of results, he was not able to revert to the last stable version that had been generated by the model.

Thus, as the primary contribution of this work, we designed, implemented, and evaluated a tool that offers a customized interface for language models like GPT, specifically targeting the construction of small web applications without writing code. This tool, called NoCodeGPT, encapsulates all prompts related to technology and architectural requirements so that the user does not need to have domain of these technical aspects. It also automatically saves the code generated by GPT in the correct folders. As a result, the user does not need to include existing code in prompts or copy GPT-generated code into local folders. Finally, it logs all interactions, prompts, and GPT responses, allowing users to easily revert to a stable version of the system in cases where the language models start producing incorrect results. In summary, with NoCodeGPT, users only write prompts related to functional requirements, while other concerns and interests are handled transparently by the proposed tool.

We also conducted two controlled experiments with the NoCodeGPT tool, in which students with limited experience in Web development were invited to use the system to develop two small Web applications (a small task management system and a question and answer forum). Despite the participants' limited experience, the results were very different from those we obtained in the first study with the standard ChatGPT interface. In the case of the first app (TodoApp), two participants implemented all four proposed user stories, while the other two participants missed just one story. In the case of the second app (ForumApp), seven participants implemented all five proposed user stories, three participants missed two stories, and one participant missed three stories. These results—entirely different from those of the first exploratory study using ChatGPT—give us confidence to assert that NoCodeGPT offers effective assistance for the construction of small Web systems by inexperienced users, without requiring the writing of any code.

The remainder of this paper is organized as follows: In Section 2, we describe our exploratory study that motivated the construction of the NoCodeGPT tool. In Section 3, we describe the features and architecture of our tool. Section 4 presents the results of using NoCodeGPT to create two web apps, the lessons learned, and also discusses threats to validity. Section 5 presents related work, and finally, Section 6 concludes.

2 | Exploratory Study

In this section, we describe the exploratory study we initially conducted to understand the potential of ChatGPT to generate Web systems without coding, that is, solely through prompts. First, we present the methodology of this study (Section 2.1), followed by its results (Section 2.2). Finally, we present the main lesson learned from the study (Section 2.3).

2.1 | Methodology

2.1.1 | Reference Implementation

To have a reference implementation for assessing the use of ChatGPT (version 4) to generate Web apps, the first author of this paper—who is an experienced software developer—implemented from scratch a simple Q&A forum, which we called appForum. The rationale was to avoid evaluating an application that was possibly used by OpenAI in the training phases of ChatGPT. We use this app as a ground-truth implementation to explore the use of ChatGPT as a no-code platform. This application implements six simple user stories:

US1: As a user, I would like to register on the forum.

US2: As a user, I would like to login on the forum.

US3: As a user, I would like to create a question.

US4: As a user, I would like to delete a question.

US5: As a user, I would like to answer a question.

US6: As a user, I would like to delete an answer.

In the back-end, we defined that the implementation should use TypeScript (programming language), Node.js with ExpressJS (server runtime), and SQLite (relational database). In the

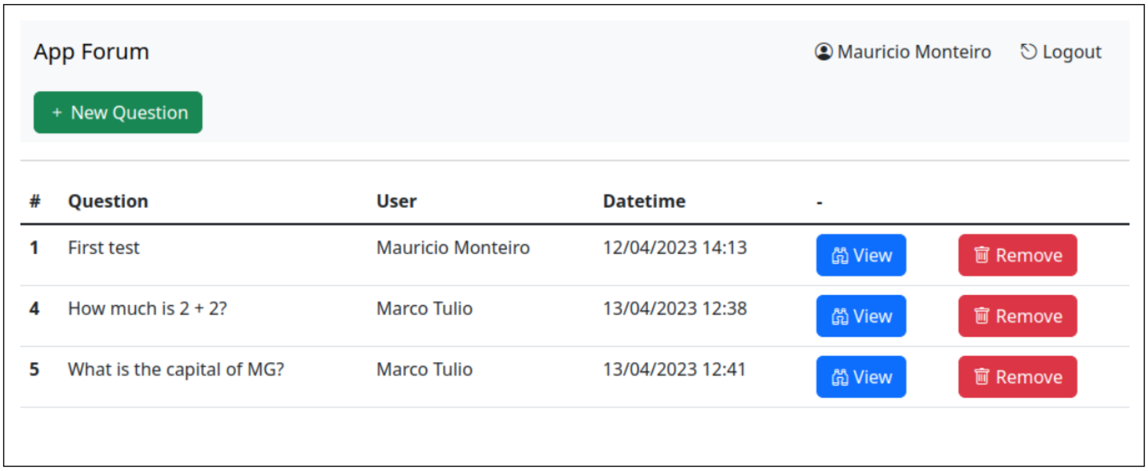

FIGURE 1 | Screenshot of the main screen (reference implementation).

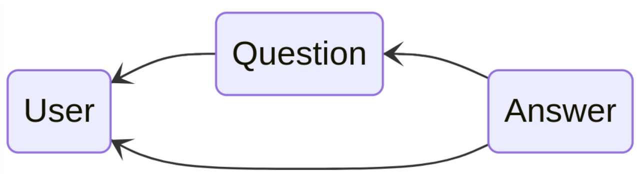

FIGURE 2 | Class diagram (back-end, reference implementation).

front-end, Vue.js and ViteJS (web frameworks) and TypeScript should be used. These technologies are widely popular and they offer several advantages that make them an attractive choice for building modern web applications. In total, the back-end implemented by the first author has 565 lines of code, three classes, and five files. The relational database has three tables (TB_USER, TB_ANSWER, and TB_QUESTION). The front-end has 820 lines of code, 11 files, six Vue.js components, and four pages. Thus, since both the backend and frontend have fewer than 1 KLOC, we claim they are representative of small web systems.

Figure 1 shows a screenshot of the main page this reference system.

Figure 2 shows a simple class diagram with the three classes of the system (USER, QUESTION, and ANSWER) and the relationship between them.

2.1.2 | Participants

In the study, we asked three developers to reimplement our reference system using ChatGPT. Participant P1 has 23 years of experience in software development. Besides being an experienced developer, he was also responsible for the reference implementation described in the previous subsection. Therefore, P1 represents the "best developer" for evaluating fChatGPT, that is, we are asking an experienced developer to use ChatGPT to generate code for an application he has implemented before. Participants P2 is a master student in Computer Science. He also has two years of experience in software development. Finally, participant P3 is an 4th year undergraduate CS student without previous professional software development experience.

Therefore, we attempted to recruit a diverse set of participants in terms of their software development experience and knowledge of the application to be developed with the support of ChatGPT. It is also worth noting that the three participants had limited experience with ChatGPT, which is expected since it is a novel technology. Particularly, they have never used ChatGPT to produce a complete software application.

2.1.3 | Inception Meeting

In this meeting, the first author presented the features, user stories, and screenshots of our appForum to both P2 and P3. Then, he asked them to rely on ChatGPT to produce code that results in an application that is as close as possible to our reference implementation and that uses the same technologies. It is important to mention that P2 and P3 had no access to the code of the reference implementation. In other words, P2 and P3 were instructed to rely only on ChatGPT to implement an identical app.

2.1.4 | Tools Used by the Participants

Participants used only a simple text editor to copy the code generated by the OpenAI tool. Interaction was exclusively through prompts, without changing parameters such as temperature, maximum token limit, or other ChatGPT settings. Essentially, we tried to reproduce the same experience that nonexperts have with language models when using the standard ChatGPT interface.

2.1.5 | Review Meeting

After using ChatGPT separately to implement our reference system, the three participants had a series of meetings to review and assess the results achieved using the AI tool. In these meetings, they presented the prompts used to interact with ChatGPT as well as executed and discussed the code generated by the tool. They also come up with the following classification for the prompts used in the study:

- Initial Prompts: prompts that describe key functional and nonfunctional requirements, as well as prompts for configuring the project and installing the necessary frameworks.
- Feature Prompts: prompts requesting the implementation of the features of the project, including prompts that request behaviors that were not supported by the generated code.
- Bug-fixing Prompts: prompts to fix bugs or incorrect behaviors in the generated code.
- Layout Prompts: prompts to style front-end elements such as buttons, tables, and text boxes.
- Other Prompts: prompts that do not fit into the previous categories, such as requesting adjustments of configurations in the developing environment.

2.2 | Results

In this section, we describe the apps constructed by each participant. First, the prompts they used are summarized in Table 1.

2.2.1 | Participant #1

This first participant started describing the user stories and technologies adopted in the app, using the prompt:

> I need to build a Web application in TypeScript with the following programming technologies: vue@latest, Express version 4, and SQLlite3 database. The back-end should follow a stateful architecture (e.g., user ids should be stored in sessions).The front-end should use the Bootstrap library. The app is a simple question and answer forum, which should implement the following user stories:
> As a user, I would like to register on the forum.
> As a user, I would like to login on the forum.
> As a user, I would like to create a question.
> As a user, I would like to delete a question.
> As a user, I would like to answer a question.
> As a user, I would like to delete an answer.

Despite listing the user stories in the initial prompt, P1 had to use 26 other prompts to request refinements in the code generated by ChatGPT, such as in the following prompt:

TABLE 1 | Prompts used by each participant.

Category	P1	P2	P3
Initial	2	1	3
Features	26	17	13
Bug Fixing	28	24	24
Layout	7	9	0
Other	2	2	6
Total	65	53	46

> Please, create a new page such that the user can view the answers for a selected question. On the main page, add a “View Answers” button that will then present this new page.

As another example, P1 had to elaborate on prompts explicitly requesting the record of new answers to a particular question. He also used other prompts to implement missing user stories and to include missing information, for example, the name of the user who answered each question.

> In the Answers Preview Screen, we should be able to register an answer to a question.

Moreover, P1 used 28 prompts to fix bugs in the code produced by ChatGPT, such as:

> There is an error in the browser console: Uncaught SyntaxError: ambiguous indirect export: “setAuthenticated.” Could you fix it?

Finally, P1 used seven layout prompts. For example, in the code initially produced by ChatGPT, the data about questions (ID, title, etc.) was presented on the page as items of a list and not as rows of a table, as planned by the participant. Thus, P1 used the following prompt to request the correct layout:

> I would like the “client/src/views/Questions.Vue” file responsible for registering questions to list the questions in a table with the question ID, the question title, and the commands to view the answers to the question and delete the question.

Figure 3 shows a screenshot of the app implemented by P1 using ChatGPT. The page in this figure is used to post answers.

2.2.2 | Participant #2

In this case, only one initial prompt was needed, as follows:

> I would like to create an user authentication form web page, with the fields “e-mail” and “password,” with two green buttons: “sign up,” which will redirect the users to a new form page where he can sign up, and “sign in,” which will authenticate existing users and redirect them to a home page, returning an error when the user is not registered. I want to use the following technologies for this: Vue.js, Express.js, TypeScript, and Sqlite. Could you help me with that, from installing these technologies to building those pages?

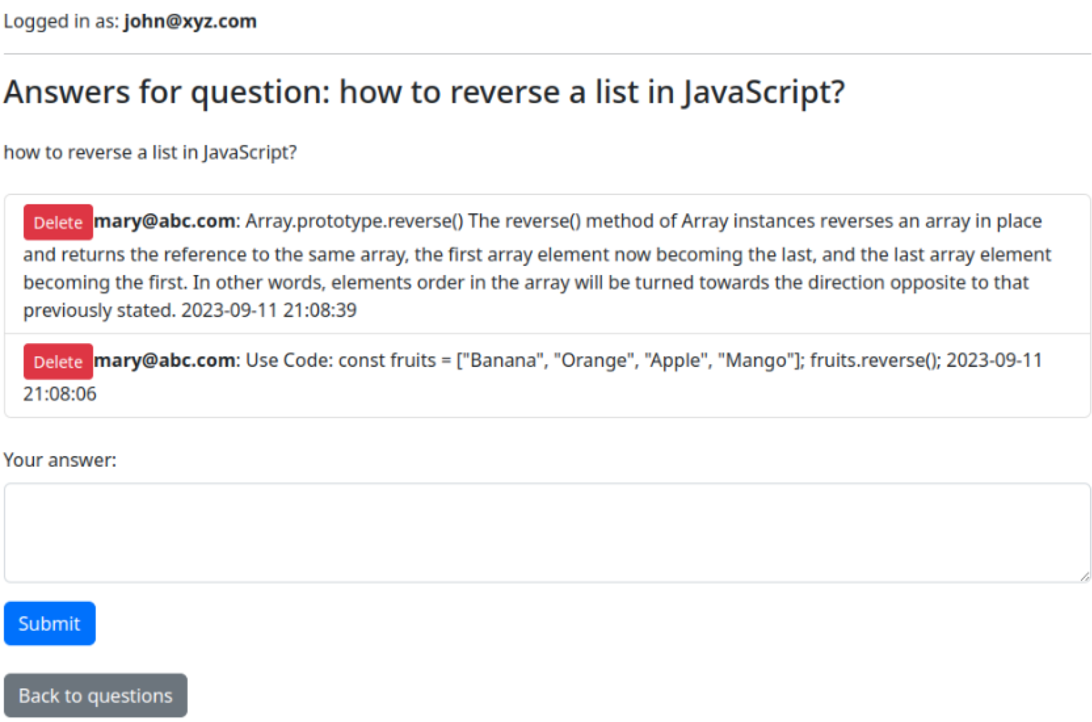

FIGURE 3 | Screenshot of the answers screen (Participant P1).

As the reader may notice, P2 started by requesting the implementation of a specific feature (authentication), by providing a high-level text describing the main fields and buttons, complemented with a request for basic error handling when the user does not exist, and then explicitly listing the desired technologies for the project. In the end, he asked ChatGPT to help in the whole process, from installing the required technologies to building the specified page.

As a response to this initial prompt, ChatGPT recommended dividing the problem into smaller steps and provided guidelines to each one (e.g., separating code by domain and indicating where newer code from an existing domain should be added). After everything was configured, ChatGPT suggested the code for the sign-up/sign-in feature.

However, P2 figured out that ChatGPT's answer was not fully correct and functional, which required him to write another 17 prompts to fix bugs in the generated code. As examples of such bugs, we can mention libraries that were actually not installed or not imported where needed, the back-end code was not correctly integrated with the front-end, and there were missing parameters on the project's configuration files.

For the remaining features, P2 changed his strategy and asked ChatGPT to first generate the front-end code with mocked objects. This first version of the front-end was then carefully tested. After that, it was integrated with the back-end code. For this last step, P2 decided to use prompts that include both the front-end and the current back-end code. Such prompts requested ChatGPT to extend the back-end with logic to handle the new features that were previously implemented and tested in the front-end. As an example of this new strategy, we have the following prompt, where P2 requests ChatGPT to generate code in the back-end (file APP.TS) to persist an answer available in a form in the front-end (file POSTDETAILS.VUE).

> Now, I want to get the post′s answers available in PostDetails.vue and when submitting the form with the answer, I want it to be saved in the database. My code from PostDetails.vue looks like that: [Source code from PostDetails.vue file], and my app.ts looks like that: [Source code from app.ts file] How can I do that?

Regarding the style prompts, P2 was able to obtain the desired style by describing how the elements should look like, for example, by informing the hexadecimal code of particular elements' colors or the shape of buttons (e.g., a button with a stadium-shaped border). He also experienced naming the colors and asking for darker or clearer tones of existing ones, which ChatGPT understood as well. Figure 4 shows a screenshot of the page to visualize questions and to provide answers, as implemented by P2 using ChatGPT.

2.2.3 | Participant #3

This participant started with the following prompt that describes the technologies and basic features of the application and asks ChatGPT to build the code:

> Please, build a forum app with the following technologies: Vue.js, Express.js, TypeScript, and Sqlite. The app should have a login screen and an option to register a user if he/she haven′t already. After logging in, the user should have the

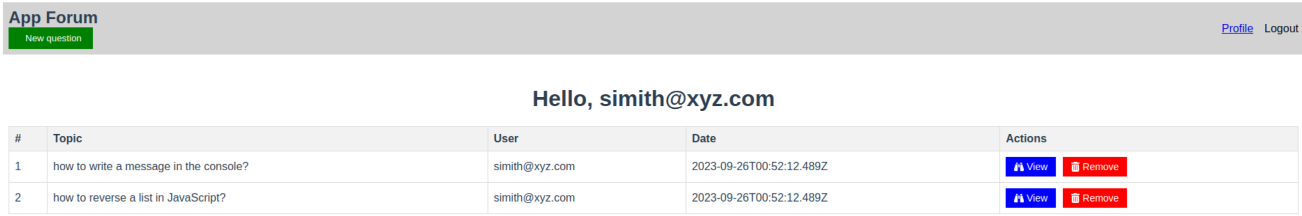

FIGURE 4 | Screenshot of the question list page (Participant P2).

> option to create a Question and to see the questions that have been created before. By clicking on the "Answer" button, a new screen will appear and the user will be able to provide an answer, save it, and then return to the main screen.

The answer to this first prompt included a good portion of the project files and also instructions to install the required technologies. However, two more prompts were necessary to obtain details about such instructions, including the following prompt:

> In topic 5 ("Set up the front-end"), how do I create views for each route?

After these initial prompts, P3 asked ChatGPT to implement other features by using 13 prompts, such as:

> Please implement code to make API calls to the back-end server for user registration, fetching questions, and saving new questions.

However, the code generated by ChatGPT missed some important files, such as index.js (in the backend). Indeed, ChatGPT informed that it was necessary to implement this file, but it does not provided the code, even after an explicit request. As a result, several bugs persisted in the generated code. Moreover, important user stories such as user registration and login were not properly implemented. The front-end for such features was created but it was not able to call the correspondent code in the back-end. ChatGPT correctly attributed this problem to an error in the connection between the front-end and back-end. However, when P3 attempted to fix the bug, ChatGPT entered in a loop, continuously suggesting previous (and also incorrect) versions of the code. After 18 attempts, P3 concluded that it was not possible to advance and he decided to quit with the project not finished.

2.3 | Lesson Learned

The main lesson learned from this study is the importance of the developer's proficiency in the technologies and frameworks used by the target system. For instance, the first two participants, who successfully completed the proposed application, were well-versed in web development and its associated technologies and frameworks. As a result, they were able to leverage their experience to formulate prompts that guided ChatGPT in fixing the bugs in the code generated by the tool.

> *Lesson Learned*: When using the standard interface provided by ChatGPT to support the end-to-end construction of web apps, experience in software development practices, architectures, and technologies is very important. This expertise is especially useful when formulating bug-fixing prompts. Therefore, it is unrealistic to expect nondevelopers to write these prompts and create a functional web app using ChatGPT.

A common bug faced by the participants occurred during the integration between the front-end and back-end. Participant 2 solved this issue using an interesting strategy. First, he generated the front-end with mocked objects and tested it thoroughly. Then, he integrated the front-end with the back-end using prompts that combined both components. These prompts instructed ChatGPT to extend the back-end to support the new features already tested in the front-end.

2.4 | Threats to Validity

There are two main threats to the validity of the results reported in this exploratory study. First, our reference application (a question and answer forum) may not represent the universe of systems that are built from scratch by software developers. However, we chose a well-known application that follows a common architecture and that uses popular technologies. Second, the code for the reference application was generated by ChatGPT using prompts formulated by only three developers. However, we carefully selected developers with different profiles (novice, moderately experienced in web development, and experienced). After the experiment, the results obtained by the inexperienced participant were clearly worse than those of the other two participants. This became evident, for example, after analyzing his prompt history and, consequently, understanding the difficulties he faced in fixing errors in the code generated by ChatGPT. Since this participant lacked knowledge of web frameworks and architectures, he was unable to formulate bug fix prompts that precisely specified which files needed modification. Additionally, he struggled with setting up the development environment and correctly configuring the framework on his local machine. On the other hand, the other two participants did not face any of these difficulties encountered by P3.

3 | NoCodeGPT: Features and Architecture

Based on the lessons learned from our exploratory study, we decided to implement a new interface for GPT, with specific features for building small web apps using only prompts, that is, without writing any code. This interface, called NoCodeGPT, can act as a replacement for the traditional interface (ChatGPT). The idea is to enable not only developers to use language models for generating web applications. Instead, we intend that users with limited software development experience could also be able to generate small web applications without having to write a single line of code.

In this section, we first describe the key features of NoCodeGPT (Section 3.1). These features emerged from our experience and observation in the Exploratory Study. Next, we also present the architecture and implementation of NoCodeGPT (Section 3.2).

3.1 | Main Features

The main functionalities of the tool are as follows:

3.1.1 | Initial Prompts

The tool defines internally the initial prompts for building web systems, meaning the prompts that specify the system's technology, architecture, and main directories, among other decisions. Thus, the user does not need to create or provide these prompts. Additionally, NoCodeGPT requests that the user provide a prompt describing the core functionality of the system under construction (see Figure 5). This prompt is important for establishing the contextual framework to be considered by the GPT model when generating the system.

3.1.2 | Predefined Features

Some features are very common in web system. Therefore, we decided to provide built-in and pretested prompts to automatically generate and include such features in the systems generated by NoCodeGPT. Currently, the tool supports two pre-defined features: a login and a user registration page.

3.1.3 | Creation and Refinement of Pages

Since NoCodeGPT is exclusively designed to build web systems, it assumes that these systems consist of a set of pages. For each page, the user must provide a name and a brief description of the desired functionalities. For example, in the case of a Q&A forum, the user might create a page called *Question Submission* with the following description: "I would like to be able to submit questions on this page. For each question, I would like the system to store the following information: …."

However, as we learned in the Exploratory Study, it is unlikely that the GPT model will generate the ideal page the user envisions in the first iteration. Consequently, NoCodeGPT includes a refinement feature where the user can request improvements to a specific page. These prompts can be of three types: new features (such as, *the page should have a button to delete a question*); bug fixes (such as *the delete question button is not working*), and layout adjustments (such as *the delete question button should be positioned right after the question's title*). Figure 6 shows an example of this functionality for the Q&A Forum. In this figure, the user is refining the *Question Submission* page.

Since refinements are restricted to one page, the tool's implementation is able to add context to the prompts provided by users. For example, our implementation automatically adds the source code files that need to be refined by the GPT model in such prompts. On the one hand, this functionality is essential to allow the tool to be used by inexperienced users, as they usually cannot correctly identify the source code files in which they intend to make a change. On the other hand, it increases the effectiveness of language models by providing them with the precise code they need to work on.

3.1.4 | Web Page Transitions

Another interesting aspect is a feature provided by NoCodeGPT to connect two pages. For instance, in our forum, the user must independently create the *Question Submission* and *Answer Submission* pages. However, afterward, they must return to the *Question Submission* page and request the creation of a button on each question that shows the page with its answers. An example is shown below:

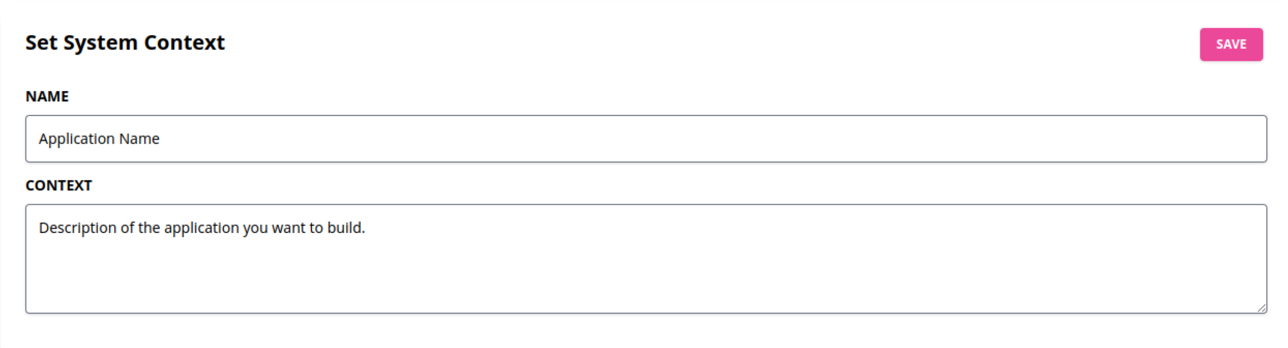

FIGURE 5 | Page to enter the name and main context of the system under development.

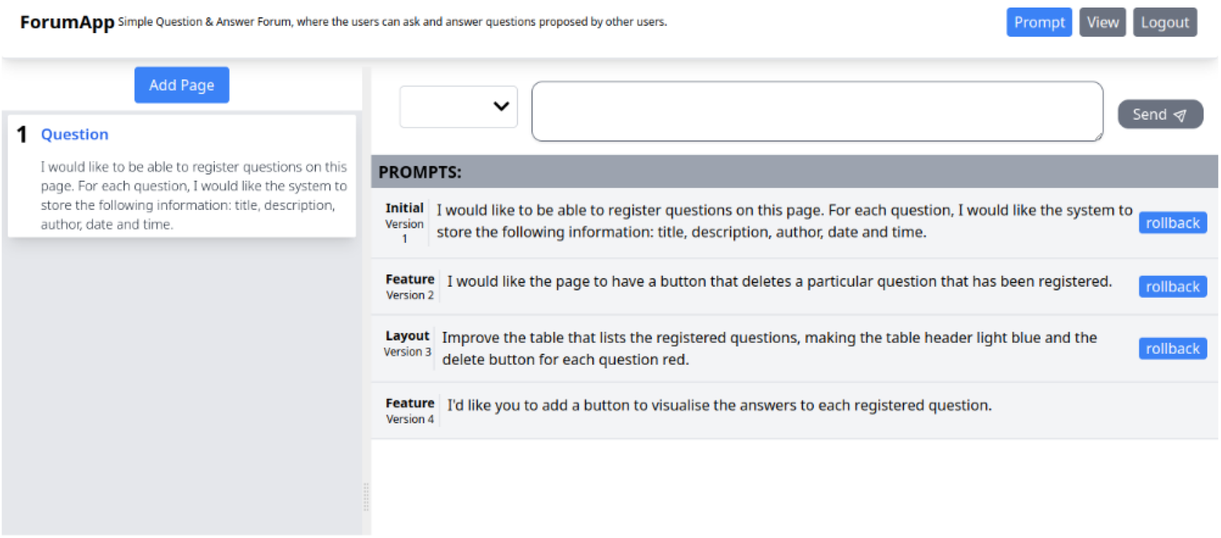

FIGURE 6 | Prompts for refining generated pages.

> Add a button to view the answers on the Answers page.

After various tests, we concluded that this alternative using simple prompts is the best solution for connecting pages. That is, the user should first create and test the pages individually. Then, they should request the connections between the pages, whenever necessary. To do this, they should simply use a prompt asking that when a certain component is clicked another page should be displayed.

3.1.5 | Version Management

NoCodeGPT also implements a simplified version control system for the code generated with the assistance of GPT. This allows users to easily restore a previous version if GPT starts generating invalid code that does not comply with the system specifications. For example, suppose the user formulates prompts requesting the implementation of features $F_1, F_2, \ldots, F_n$, with each submission generating a respective version V_i of the system. However, in a particular version V_j, the language model might start hallucinating and generate invalid code. In such cases, the user can revert to version V_{j-1} using a single button provided in NoCodeGPT's interface.

This feature was specifically designed to address a difficulty encountered by participant P3 in our Exploratory Study. This participant abandoned the study because ChatGPT chose a path that resulted in successive invalid versions of the system. Moreover, using the standard ChatGPT interface, backtracking to the last correct version is not easy. The user has to save these versions manually, which is not a natural procedure for less experienced users.

3.1.6 | Execution and Visualization

NoCodeGPT also provides a button that allows the user to quickly run and check the behavior of the code generated by GPT. This way, the user does not need to have knowledge of command-line tools, such as compilers and interpreters.

3.2 | Architecture and Implementation

The implementation of NoCodeGPT has two modules: front-end and back-end, as shown in Figure 7. The front-end is responsible for controlling the functionality of the tool's pages and for guiding the user through the workflow to build the Web app. The back-end is responsible for receiving requests sent by the front-end and processing the necessary information to build the prompts that are submitted to the OpenAI API. The back-end also stores all interactions with the OpenAI platform in a database.

Specifically, the front-end implementation uses the following technologies: Vue.js (version 3) with TypeScript using the Composition API, HTML, and CSS. For the back-end implementation, ExpressJS, SQLite 3, and JSON Web Token are used. The front-end has 1,847 lines of code, and the back-end has 3,342 lines, totaling 5,189 lines.

Figure 8 provides a detailed view of the architecture of the implemented tool. The front-end consists of the Main class, which controls the initialization of this component, and the App class, which manages the interface logic. Page route management is handled by the Route class, which is capable of displaying any of the pages, such as AddPageView and LoginView. Specifically, the application pages are implementations of Vue Single-File Components. These components call the endpoints implemented in the back-end using the CallService class, which relies on the standardized implementation of the FetchAPI class.

In the back-end, the prompts are initially processed by adding other important information to obtain more accurate responses, and, ultimately, the GPT API is called to generate the code requested by users. The PromptRequest and PromptResponse classes store, respectively, the requests and responses made to this API. Detailed information from the previous requests is also stored to implement the version control

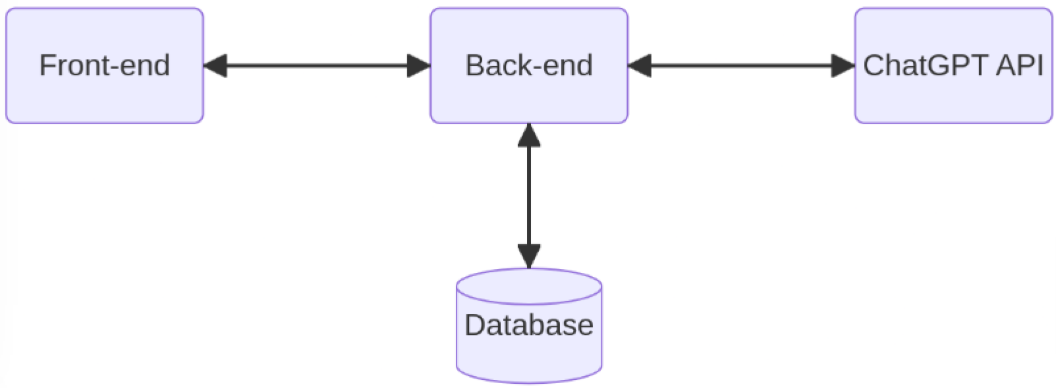

FIGURE 7 | NoCodeGPT's high-level architecture.

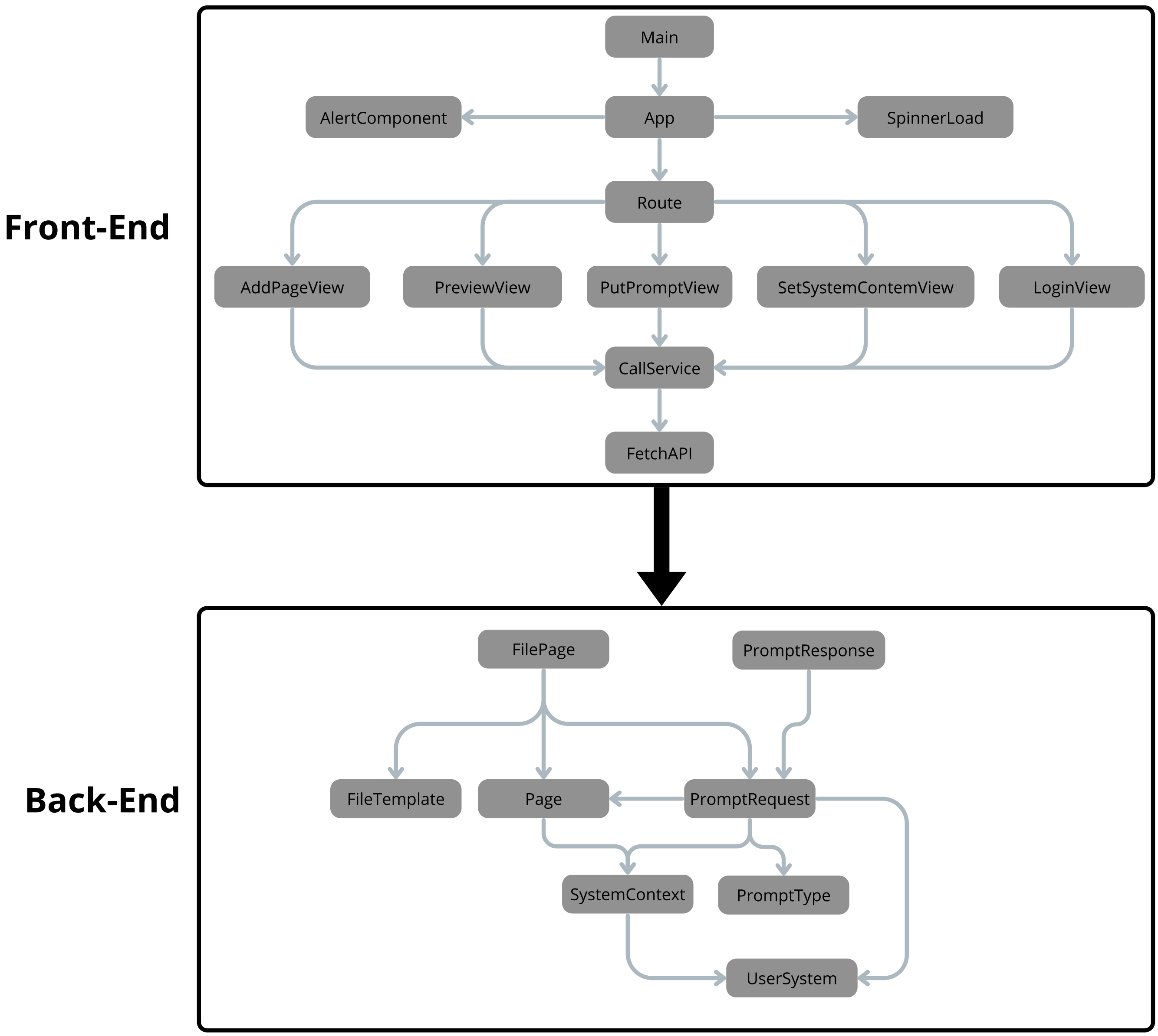

FIGURE 8 | Front-end and Back-end architecture.

mechanism provided by the tool. Finally, the PromptType class stores the templates of the prompts sent to the OpenAI platform.

3.2.1 | Temperature

From the experience of the Exploratory Study, we concluded that lowering the temperature in fact makes the model more deterministic and also accurate. Therefore, we decided to reduce the temperature used in NoCodeGPT to zero.

4 | NoCodeGPT Evaluation

In this section, we present an evaluation of NoCodeGPT, conducted in two phases, with a total of 14 participants, who were invited to implement two small web applications using only prompts and our tool. Our main objective is to verify whether NoCodeGPT helps to overcome the limitations identified in the Exploratory Study.

4.1 | Methodology

The evaluation of the proposed tool was conducted in two phases. First, a pilot test was carried out with four participants. The goal was to test and validate the implementation of the proposed tool with a small number of participants and assess its feasibility. Specifically, the participants used the tool to implement a simple task management application with the following user stories:

1. As a user, I want to add a new task to the list.
2. As a user, I want to edit an existing task.
3. As a user, I want to mark a task as completed.
4. As a user, I want to remove a task from the list.

The participants received initial training on the tool (30 min) and were also provided with a low-fidelity mockup of the application.

TABLE 2 | Participants' experience on web development.

Years of experience	TodoApp	ForumApp
No experience	2	8
Less than one year of experience	2	2

We decided to provide this mockup so that participants could have an idea of the functionality they needed to implement. However, we clarified that the mockup would serve only as an example, meaning the application to be created did not need to have an identical layout.

Following this pilot experiment, we conducted a larger experiment with 10 participants. In this case, we used the ForumApp from our Exploratory Study (Section 2) as the target application.[3] The participants also received initial training and a mockup of the requested app. They were asked to implement the following user stories:

1. As a user, I want to add a new question to the forum.
2. As a user, I want to delete a question from the forum.
3. As a user, I want to access the answers to a given question on a separate page.
4. As a user, I want to answer a question in the forum.
5. As a user, I want to delete an answer.

Compared to the Exploratory Study, we made two important changes to the list of user stories of ForumApp. First, we removed the stories related to login and user registration, as these are predefined features in NoCodeGPT, as explained in Section 3.1.2. Second, we added a story explicitly stating that, given a question, it is important to access a second page with its answers. This was done to prevent users from implementing the entire app on a single page. If that happened, we would not be able to evaluate NoCodeGPT's ability to support the creation of apps composed of multiple web pages.

Table 2 summarizes the participants' web development experience in each experiment (TodoApp and ForumApp). All participants are undergraduate Computer Engineering students in their first or second year. Additionally, as shown, they match the ideal user profile for NoCodeGPT, meaning they have either no experience or, at most, one year of experience in web development.

4.2 | Pilot Experiment Results (Todoapp)

In the pilot experiment, two participants managed to build the TodoApp application with all four proposed stories. However, two participants did not implement one of the stories. Figure 9 shows

FIGURE 9 | Screenshot of participant P2's application.

a screenshot of an application built by one of the participants who successfully completed all four stories.

Moreover, Table 3 details the stories that were implemented by the participants. As we can see, participant P4 was unable to implement task editing, while participant P1 was unable to implement a feature to mark a task as completed. In both cases, the participants attempted to implement the functionality, but the code generated by GPT did not work as expected. Thus, they decided to give up without attempting again. On the other hand, the creation of new tasks, as well as their removal, was implemented by all four participants.

Figure 10 shows the number of prompts each participant used to build the TodoApp. Participants P2 and P3 interacted the most with the tool and were the only ones to implement all user stories, each using 16 prompts. In contrast, participants P1 and P4 used nine and eleven prompts, respectively. The most common prompt type was for requesting features, except for participant P2, who used eight prompts to improve the app's layout. For P1, there was an even split between feature and bug-fix prompts, with four prompts each. Interestingly, P1 and P4 did not use any layout prompts, indicating they were satisfied with the initial layout proposed by GPT. Lastly, all participants used exactly one initial prompt.

TABLE 3 | Participants who implemented each user story in the pilot experiment (TodoApp).

User stories	Participants who succeeded
As a user, I want to add a new task	P1, P2, P3, P4
As a user, I want to edit an existing task	P1, P2, P3
As a user, I want to mark a task as completed	P2, P3, P4
As a user, I want to remove a task from the list	P1, P2, P3, P4

NoCodeGPT's version control functionality was heavily used during the pilot experiment. The four participants carried out a total of 16 rollbacks while building the proposed web application. Table 4 shows that participant P4 made the most use of this feature, requesting five rollbacks.

Figure 11 illustrates the rollback operations carried out by participant P2. In this figure, each branch represents a sequence of prompts that were abandoned (i.e., that were concluded with a rollback). Specifically, P2 performed four rollbacks, after prompts 4, 6, 10, and 13. We can also see that P2 used 16 prompts in total, but seven prompts were discarded. Only nine prompts resulted in usable code (prompts 1-2, 7-9, 11, 14-16).

4.3 | Results of the Second Experiment (Forumapp)

In the second experiment (ForumApp), seven participants successfully implemented all five stories. The last three participants failed to implement at least two user stories. Figures 12 and 13 show screenshots of the application created by one of the participants who completed all five stories. Figure 12 shows the question registration page, and Figure 13 shows the answer registration page.

TABLE 4 | Rollbacks by participants in the pilot experiment (TodoApp).

Participant	P1	P2	P3	P4
Rollbacks	4	4	3	5

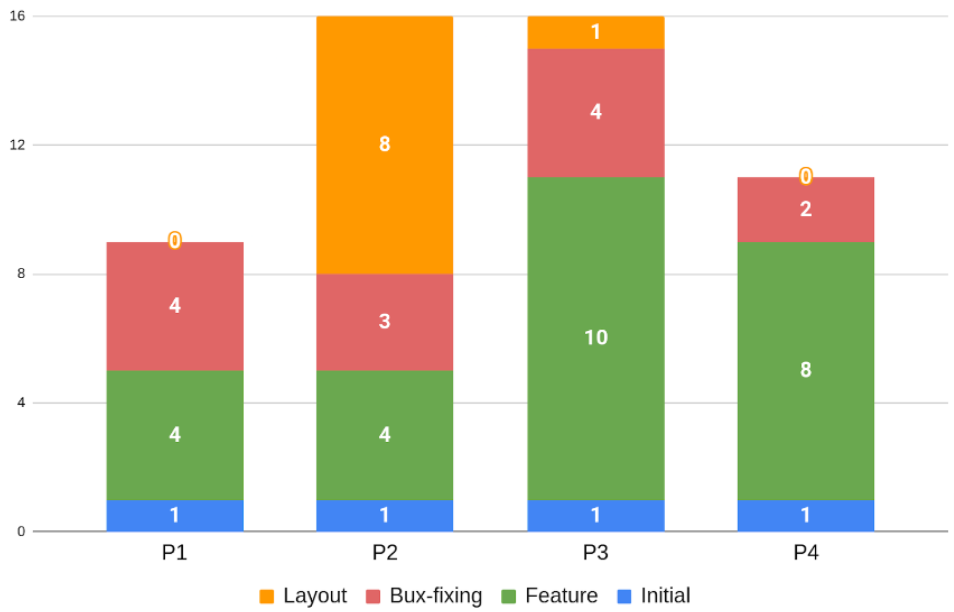

FIGURE 10 | Types of prompts implemented by each participant in the pilot experiment (TodoApp).

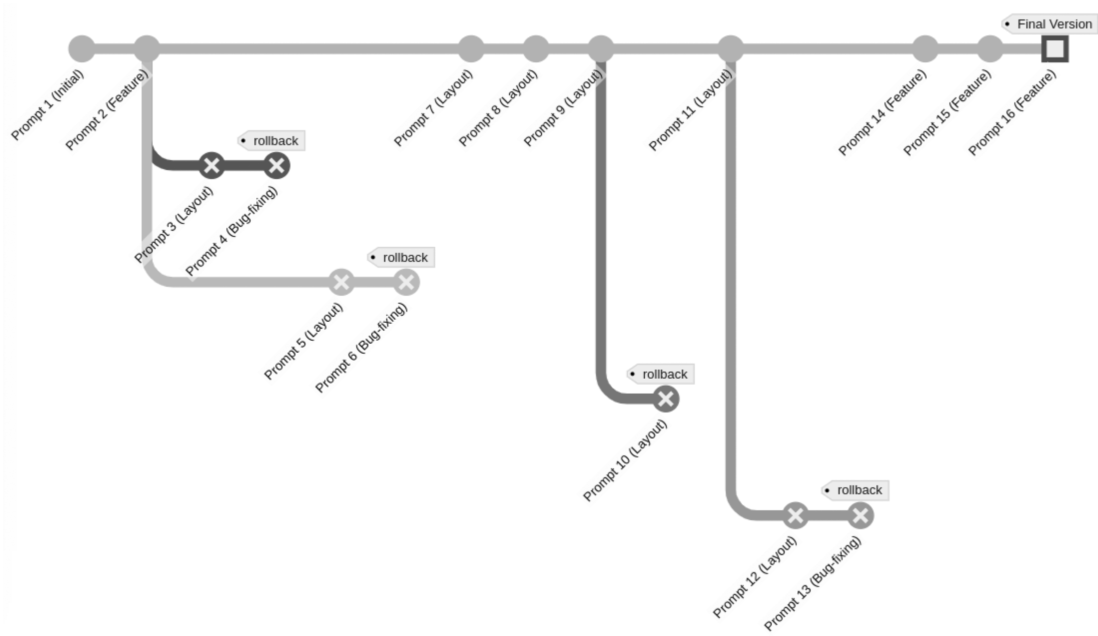

FIGURE 11 | Rollbacks performed by participant P2 in the pilot experiment (TodoApp).

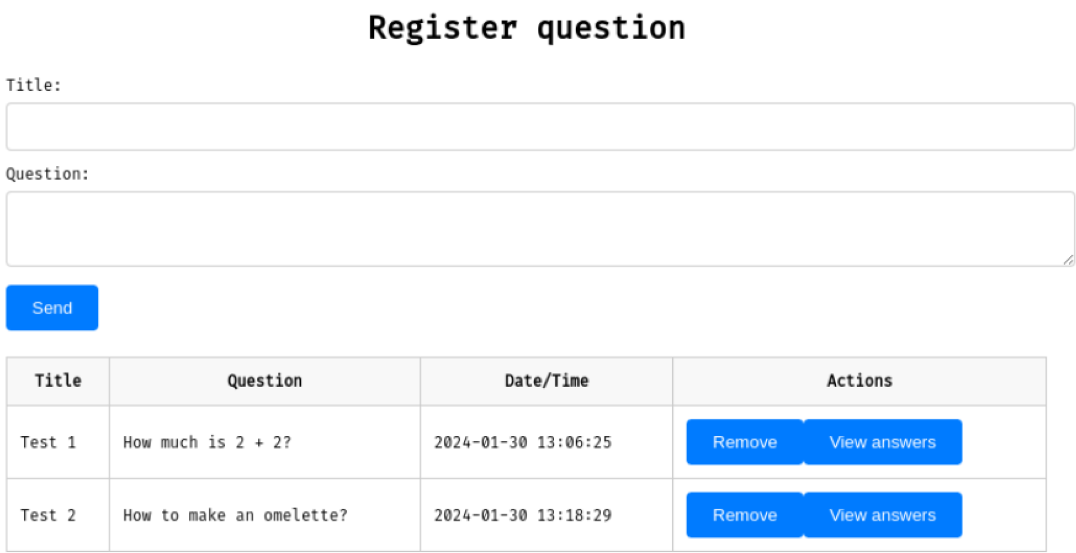

FIGURE 12 | Question registration page created by participant P7 in the second experiment.

Table 5 shows the stories implemented by the participants in the second experiment. As we can see, participants P8, P9, and P10 were unable to implement the stories about answering questions and deleting answers. Participant P9 was the only one who failed to implement the third story, which defines that from the question page it should be possible to access the answers pages. We also compared the user story completion rates from this experiment with those from the second experiment using a t-test. We concluded that there is no statistically significant difference between the two distributions (p-value = 0.4667).

Figure 14 shows the number of prompts used by participants to build the ForumApp. Feature-type prompts were the most frequently used by participants, totaling 37 prompts, followed by 20 initial-type prompts. Each participant used two initial-type prompts: one for the page for registering questions and another for the page for registering answers. The bug-fixing and layout categories have 14 prompts each.

Of the seven participants who successfully implemented the application, P7 used the highest number of prompts (12), while P3 used the lowest one (5). The main reason for this difference seems to be the level of detail in their initial prompts. P3 started with a more detailed prompt, as presented next:

> Create a page where I can register questions, where the registration includes a title, the text of the question, the logged-in

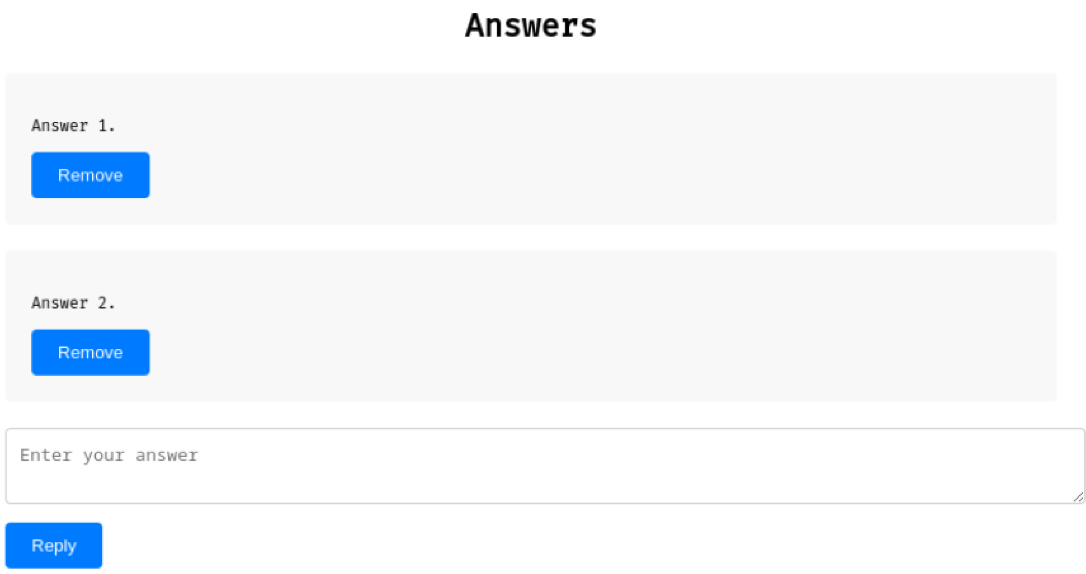

FIGURE 13 | Answer registration page created by participant P7 in the second experiment.

TABLE 5 | Participants who implemented each user story in the second experiment.

User stories	Participants who succeeded
As a user, I want to add a new question	All participants
As a user, I want to remove an existing question	All participants
As a user, I want to access the answers page	All participants, except P9
As a user, I want to answer a question	All participants, except P8, P9, P10
As a user, I want to delete an answer	All participants, except P8, P9, P10

> user who asked the question, and the time the user asked. The questions should be listed in a table showing the title, the question text, the user who asked, and the date/time, alongside a button to delete the question and a button to view answers. The view answers button should go to an answer registration page for the selected question.

On the other hand, P7's initial prompt was more succinct as follows:

> Create a question registration page, which will have a box to enter the title of the question and add the description of the question. The questions will be listed separately in a table that will show the title of the question, the user who registered it, and the date and time.

Participants P9 and P10 were the only ones who did not use layout-type prompts. This behavior was common among participants who struggle to finalize the application. Since both participants were unable to implement all functionalities, they did not reach the stage where layout-type prompts are typically used, i.e., after everything is working as expected. Participants P1, P2, and P3 were the only ones who did not use bug-fixing prompts, likely because their more detailed initial prompts led to more precise results by the GPT model.

Finally, Table 6 shows the number of rollbacks by each participant. As we can see, this feature was used less during the development of ForumApp compared to the pilot study with TodoApp. Our hypothesis is that, although ForumApp has two pages, they are simpler than the single page specified in TodoApp. These simpler pages made it easier for participants to define more effective prompts. As a result, GPT was able to generate correct code on the first attempts, reducing the need for rollbacks.

4.4 | Participants' Perceptions

After the experiment, the participants also reported their perceptions about NoCodeGPT in a simple form. They answered two questions:

4.4.1 | What Are the Tool'S Most Positive Points?

Among the positive points highlighted by the participants were the simplicity of NoCodeGPT's interface (10 participants), the convenience of not having to copy-and-paste code (9 participants), and the lack of need for prior knowledge of programming languages (10 participants). Below are some of the positive aspects of the tool mentioned by the participants:

We were able to create a functional application without using any code, just simple instructions. The possibility of selecting previous versions was fundamental for correcting some bugs.

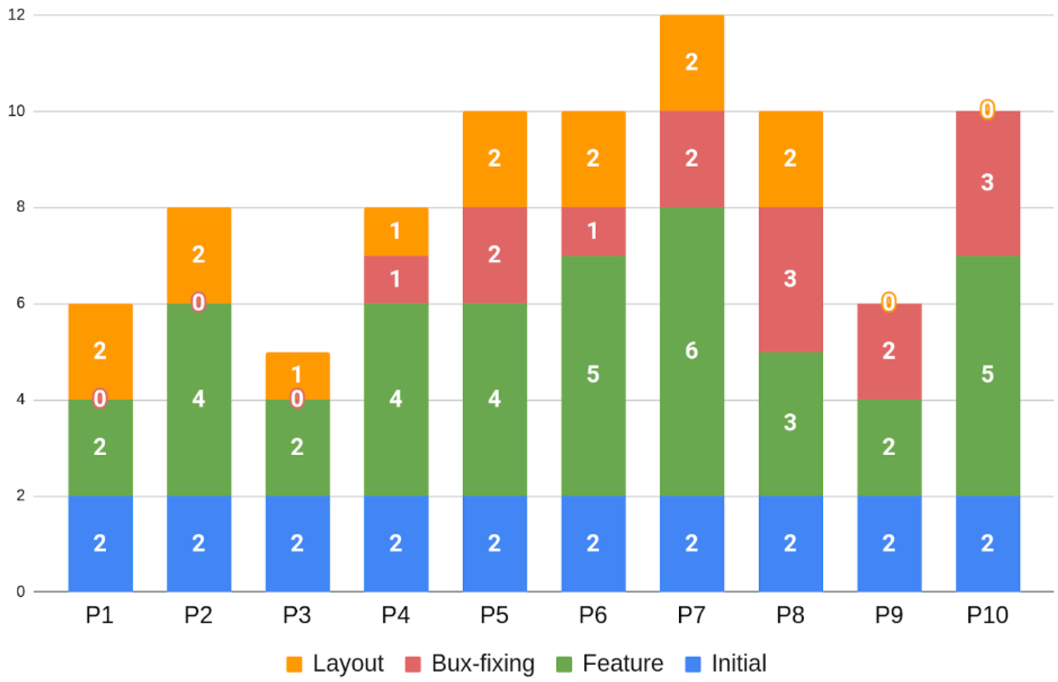

FIGURE 14 | Total prompts used by each participant stacked by type.

TABLE 6 | Rollbacks by participants in the second experiment.

Participant	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
Rollbacks	0	0	0	0	0	1	2	0	0	2

NoCodeGPT has an interesting purpose and is very useful, especially if you have a minimum knowledge of development. One of the best features is being able to go back to previous versions.

The tool is very easy to use; and the feature of not having to copy-and-paste any code or have prior knowledge of programming languages (although knowledge certainly increases is benefit); The feature to go back a version is very useful and helps a lot in the construction.

4.4.2 | What Are the Downsides of the Tool?

The main negative point pointed out by participants was the delay in responses during interactions with the prompts (14 participants), as commented by the following participants:

NoCodeGPT at times takes too long to respond. Improve the response time.

Slow responses, I suggest increasing the processing speed of the tool.

It takes too long to execute the commands.

However, it is important to note that this delay refers to accessing the GPT API provided by OpenAI. Thus, it is a variable over which we have no control.

4.5 | Lessons Learned

In the exploratory study, we concluded that the default interface offered by ChatGPT is not suitable for users without programming experience. Therefore, we decided to invest in the design and implementation of NoCodeGPT, a wrapper for the GPT API with features that make it easier for nonexpert users to create a simple web application without writing code.

In this section, we reported the results of two studies designed to evaluate the effectiveness of NoCodeGPT in accomplishing its purpose. In the first study (TodoApp), two participants implemented all four proposed user stories, while the other two participants missed just one story. In the second study (ForumApp), seven participants implemented all five proposed user stories, three participants missed two stories, and one participant missed three stories.

> *Lesson Learned*: The results obtained with both apps showed that NoCodeGPT is effective in helping inexperienced users implement simple web applications without having to write code.
>
> - In the exploratory study, using the default interface provided by ChatGPT, an inexperienced participant was unable to implement any of the proposed user stories.

- In the experiments described in this section, using the customized interface provided by NoCodeGPT, a scenario of total failure did not occur with any of the participants. Considering both experiments, NoCodeGPT was used by 14 inexperienced participants. More than half of such participants (9 participants) successfully completed the proposed applications, while the others completed at least half of the proposed user stories.

Next, we briefly discuss the role and contribution that the key features of NoCodeGPT had in these results.

- NoCodeGPT establishes a clear distinction between two categories of prompts: (1) prompts that define technologies and architectures; and (2) prompts that define functional requirements. The latter prompts are handled and encapsulated by the tool. Consequently, users are solely responsible for writing prompts describing functional requirements.
- NoCodeGPT also manages and stores the code generated by GPT. Thus, users do not need to copy and paste the generated code to a local directory or create a specific folder structure for each application and architecture. These concerns are completely automated by the proposed tool. It also automates related tasks, such as installing libraries and configuring environment variables. It is worth mentioning that such tasks represented major obstacles for the nonexpert participant from our Exploratory Study (Section 2).
- The rollback feature has been of major importance in NoCodeGPT's performance, particularly in the experiment with the TodoApp. In essence, this feature reduces the chances that users give up when a bug recurrently appears in the generated code. When this happens, users can readily restore a version that not have this bug.
- The code execution and visualization button was also very important for users to quickly check and identify issues in the generated code, including both bugs and layout problems.

Although NoCodeGPT was designed to help developers with limited familiarity with web software development, we believe it can also assist more experienced developers, particularly in prototyping activities or when implementing Minimum Viable Products (MVPs).

4.6 | Threats to Validity

The results presented in this section are vulnerable to two main threats to validity. First, the limited sample size of 14 participants who formulated prompts for the construction of the applications. It is possible that these participants do not represent the entire population of users who intend to use ChatGPT to support end-to-end software construction. However, participants with a similar profile and level of experience to that of P3 from the exploratory study were carefully selected to mitigate this limitation. Secondly, ForumApp and TodoApp may not represent the full spectrum of systems that are built from scratch by software developers. However, they follow a common architecture (web-based, with front-end and back-end components) and use popular technologies such as TypeScript, Vue.js, and SQLite, which enhances the generalizability of our findings.

5 | Related Work

In this section, we discuss recent papers related to our study. First, we comment on papers related to our central goal of using LLMs to support software construction. After, we discuss papers that rely on LLMs to automate software maintenance tasks, including fixing bugs and writing tests.

5.1 | Software Construction

Researchers from Microsoft Research, GitHub, and MIT conducted a controlled experiment with professional developers who were asked to implement an HTTP server in JavaScript [12]. The treatment group, which had access to GitHub Copilot, was able to complete the proposed task 55.8% faster. However, we claim that our study uses an application that reflects the practice of software development today. For example, our Q&A forum includes a front-end (implemented in a widely used framework, Vue.js), an API provided by the back-end (in the form of a set of HTTP end-points), and a SQLite database.

Le and Zhang evaluated the use of ChatGPT in a very specific context, that is, to automatically implement parsers for log files [13]. In a dataset with thousands of log files, the authors report that ChatGPT achieved 71% accuracy, that is, log messages that were correctly recovered by the parser generated by the tool (from a generic prompt). White and colleagues describe a set of 13 prompt patterns for solving a wide range of software engineering problems, from system design to implementation and maintenance [14]. However, these prompts are more abstract and generic than the ones we use in our study. Specifically, our prompts aim to generate usable and runnable software from agile-based requirements (written as user stories). White and colleagues also propose six prompt patterns related to code maintenance, evolution, and refactoring tasks, which are outside the scope of our research. Finally, it is also important to mention that in our work, we not only define the prompts but also use them in a real-world context for end-to-end software construction.

Similarly, Sadik and colleagues from the Honda Research Institute comprehensively describe various applications of LLMs in software engineering, including code generation, documentation, bug detection, and refactoring [15]. However, they do not apply their prompts in real-world contexts. Researchers from Meta describe a competitor to GitHub Copilot they are developing internally at the company [16]. In addition to describing the design and implementation of the system, called CodeCompose, the authors present results regarding the tool's usage and adoption. They conclude by mentioning that "in addition to assisting in code implementation (authoring), CodeCompose is introducing other positive effects, such as encouraging developers to generate more documentation, helping them discover new APIs, etc."

Nguyen and Nadi evaluated the use of GitHub Copilot on a dataset of programming problems (LeetCode) [10]. The correction of the responses generated by Copilot was assessed using the dataset's own test suite. Additionally, the authors evaluated code quality metrics, such as cyclomatic complexity. Mastropaolo and colleagues assessed the robustness of the code generated by GitHub Copilot [17]. Their comparison involved two scenarios: code generation from JavaDoc comments and a semantically modified version of such comments. In a sample of 892 methods from Java projects, the recommendations generated by ChatGPT differed in nearly half of the tests, demonstrating the tool's sensitivity to the prompts provided as input.

In a recent blog post, Guo explored the advantages and limitations of ChatGPT in complex programming tasks [18]. As in our study, he highlights the continued need for human guidance and the need for expertise and previous education in Software Engineering for an effective use of the tool. Dakhel and colleagues examine GitHub Copilot's efficacy in generating solutions for core Computer Science problems [9]. They compare Copilot's solutions with those produced by human programmers and find that, while Copilot generates solutions for most problems, they exhibit more bugs compared to human solutions.

5.2 | Software Maintenance and Testing

Sobani and colleagues evaluated the performance of ChatGPT in bug fixing, using a dataset commonly adopted in this field (QuixBugs) [1]. The authors conclude that the system's performance is considerably better than other approaches proposed in the literature. Specifically, ChatGPT was able to repair 31 out of 40 bugs evaluated in the research. Siddiq and colleagues, on the other hand, report less promising results regarding the use of ChatGPT to implement unit tests [3]. In fact, in a first dataset, the statement coverage of the automatically generated tests was good (80%). However, in a second dataset, the results were worse (only 2% coverage). The authors also report that the generated tests have some smells, such as Duplicate Asserts and Empty Tests.

Asare, Nagappan, and Asokan evaluate whether the code generated by GitHub Copilot has the same security flaws as code written by developers [19]. The conclusion was that in 33% of cases, Copilot essentially replicates the vulnerabilities that exist in a dataset of C and C++ systems. Pearce and colleagues explore the security implications of code generated by GitHub Copilot [20]. The authors evaluate Copilot's performance across three dimensions of code generation, specifically assessing its capabilities in addressing various weaknesses, prompts, and domains. Their analysis covers 1,692 programs generated by Copilot, revealing that approximately 40% of them exhibit vulnerabilities.

Meta's software engineers conducted a study employing the TestGen-LLM tool [21]. This tool automatically enhances pre-existing human-written tests for Instagram and Facebook platforms using Language Model Models. The results showed that 11.5% of all test classes extended by TestGen-LLM had improved tests. Moreover, 73% of the recommendations generated by TestGen-LLM were accepted by Meta's software engineers for implementation in the production environment.

Almeida et al. [8] explore the use of ChatGPT to assist with API migration. Particularly, they use ChatGPT to migrate a client application to a newer version of SQLAlchemy, a popular Python ORM. Three prompt types—Zero-Shot, One-Shot, and Chain of Thoughts—are evaluated, with One-Shot yielding the best results. Using this type of prompt, all columns were successfully migrated, and the application was upgraded to leverage new SQLAlchemy features like `asyncio` and typing, while maintaining its original behavior. Silva et al. [7] assessed ChatGPT's effectiveness in detecting code smells in Java projects, focusing on Blob, Data Class, Feature Envy, and Long Method across three severity levels. Two prompts—a generic one and a specific one—are tested, with results showing that the specific prompt improves accuracy by 2.54 times. ChatGPT performs better on critical smells (F-measure = 0.52) than minor ones (F-measure = 0.43).

6 | Conclusion

Language models are being widely used by software engineers. Therefore, in this article, we started by investigating the use of ChatGPT for generating small Web applications by users with limited experience in software development. We concluded that the standard GPT interface was not designed to specifically address this task. For this reason, we proposed, designed, and implemented a new interface for using language models, which provides a more user-friendly experience for building Web apps. This new interface, which we called NoCodeGPT, encapsulates the entire exchange of prompts with the AI model. As a result, users no longer need to write prompts related to technologies and architecture patterns or copy files to their respective local folders. We also conducted a controlled experiment with 14 students who are beginners in Web development. The results provided convincing evidence that the proposed tool is indeed superior to the standard ChatGPT interface, helping inexperienced developers build small Web apps without writing code. For example, more than half of the participants (9 out of 14) successfully completed the proposed applications, while the others completed at least half of the proposed user stories.

Currently, the evolution of AI models is progressing at an impressive speed, which may raise questions about the longevity of wrappers like NoCodeGPT. However, we believe that our main contribution is demonstrating that AI-assisted software development requires certain fundamental tasks, such as defining an architecture, storing files in the correct directories prescribed by that architecture, running the system for testing, and quickly reverting to previous versions in case of model hallucinations. These concerns and tasks remain independent of AI model advancements, ensuring they will continue to be relevant in the end-to-end development of software apps supported by language models.

As future work, we intend to evaluate the use of our interface with more apps and more developers. We also plan to make it compatible with other language models, including open-source models. We also aim to investigate the construction of new types of applications, such as mobile apps. Finally, we also plan to investigate other mechanisms for controlling and preventing hallucinations besides the rollback mechanism currently implemented by NoCodeGPT.

Author Contributions

Mauricio Monteiro: conceptualization; data curation; investigation; methodology; validation; writing – original draft; writing – reviewing and editing. **Bruno Castelo Branco:** conceptualization; methodology; validation. **Samuel Silvestre:** conceptualization; methodology; validation. **Guilherme Avelino:** conceptualization; methodology; validation; writing – reviewing and editing. **Marco Tulio Valente:** conceptualization; methodology; validation; writing – original draft; writing – reviewing and editing.

Acknowledgments

Our research is supported by grants from CNPq and FAPEMIG. The Article Processing Charge for the publication of this research was funded by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) (ROR identifier: 00x0ma614).

Conflicts of Interest

The authors declare no conflicts of interest.

Data Availability Statement

The code of our NoCodeGPT tool is available at: https://github.com/mauricioms/nocodegpt. This repository also contains the prompts from the exploratory study and the data on the evaluation of NoCodeGPT.

Endnotes

[1] https://github.blog/2023-06-27-the-economic-impact-of-the-ai-powered-developer-lifecycle-and-lessons-from-github-copilot/.

[2] https://www.youtube.com/live/outcGtbnMuQ.

[3] When selecting these applications, our focus has always been on simple web systems, but with an architecture that resembles real-world systems—that is, with a frontend built using frameworks like Vue.js, a backend providing a Web API, and a relational database. Additionally, the applications should implement a small number of user stories (e.g., four or five stories).

References

1. D. Sobania, M. Briesch, C. Hanna, and J. Petke, "An Analysis of the Automatic Bug Fixing Performance of ChatGPT," in *Proceedings of the IEEE/ACM International Workshop on Automated Program Repair (APR@ICSE 2023)* (IEEE, 2023), 23–30.

2. C. S. Xia, Y. Wei, and L. Zhang, "Automated Program Repair in the Era of Large Pre-Trained Language Models," in *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering* (Springer, 2023), 1482–1494.

3. M. L. Siddiq, J. C. S. Santos, R. H. Tanvir, N. Ulfat, F. Al Rifat, and V. C. Lopes, "Exploring the Effectiveness of Large Language Models in Generating Unit Tests," *CoRR* (2023): 1–24, abs/2305.00418.

4. M. Geng, S. Wang, D. Dong, et al., "Large Language Models are Few-Shot Summarizers: Multi-Intent Comment Generation via in-Context Learning," (2024).

5. J. Shin, C. Tang, T. Mohati, M. Nayebi, S. Wang, and H. Hemmati, "Prompt Engineering or Fine Tuning: An Empirical Assessment of Large Language Models in Automated Software Engineering Tasks," 2023.

6. J. Li, L. Li, J. Liu, Y. Xiao, X. Liu, and J. W. Keung, "Large Language Model ChatGPT Versus Small Deep Learning Models for Self-Admitted Technical Debt Detection: Why Not Together?," *Software: Practice and Experience* 55, no. 1 (2025): 3–28.

7. L. L. Silva, J. Silva, J. E. Montandon, M. Andrade, and M. T. Valente, "Detecting Code Smells Using ChatGPT: Initial Insights," in *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (John Wiley & Sons, 2024), 1–7.

8. A. Almeida, L. Xavier, and M. T. Valente, "Automatic Library Migration Using Large Language Models: First Results," in *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (IEEE, 2024), 1–7.

9. A. M. Dakhel, V. Majdinasab, A. Nikanjam, et al., "Github Copilot Ai Pair Programmer: Asset or Liability?," *Journal of Systems and Software* 203 (2023): 111734.

10. N. Nguyen and S. Nadi, "An Empirical Evaluation of GitHub Copilot's Code Suggestions," in *Proceedings of the 19th International Conference on Mining Software Repositories (MSR)* (Elsevier, 2022), 1–5.

11. M. T. Valente, *Software Engineering: A Modern Approach* (ACM, 2024).

12. S. Peng, E. Kalliamvakou, P. Cihon, and M. Demirer, "The Impact of AI on Developer Productivity: Evidence From GitHub Copilot," *CoRR* abs/2302.06590 (2023): 754–768.

13. V.-H. Le and H. Zhang, "An Evaluation of Log Parsing With ChatGPT," *CoRR* abs/2306.01590 (2023): 1–19.

14. J. White, S. Hays, Q. Fu, J. Spencer-Smith, and D. C. Schmidt, "ChatGPT Prompt Patterns for Improving Code Quality, Refactoring, Requirements Elicitation, and Software Design," 2023.

15. A. R. Sadik, A. Ceravola, F. Joublin, and J. Patra, "Analysis of ChatGPT on Source Code," *CoRR* abs/2306.00597 (2023): 1–13.

16. V. Murali, C. S. Maddila, I. Ahmad, et al., "CodeCompose: A Large-Scale Industrial Deployment of AI-Assisted Code Authoring," *CoRR* abs/2305.12050 (2023): 1–11.

17. A. Mastropaolo, L. Pascarella, E. Guglielmi, et al., "On the Robustness of Code Generation Techniques: An Empirical Study on GitHub Copilot," in *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering* (ACM, 2023), 2149–2160.

18. P. Guo, "Real-Real-World Programming With ChatGPT," (2023), https://www.oreilly.com/radar/real-real-world-programming-with-chatgpt/.

19. O. Asare, M. Nagappan, and N. Asokan, "Is Github's Copilot as Bad as Humans at Introducing Vulnerabilities in Code?," *Empiririal Software Engineering* 28, no. 6 (2023): 129.

20. H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions," in *Proceedings of the 43rd IEEE Symposium on Security and Privacy* (IEEE, 2022), 754–768.

21. N. Alshahwan, J. Chheda, A. Finegenova, et al., "Automated Unit Test Improvement Using Large Language Models at Meta," *CoRR* (2024): 1482–1494, abs/2402.09171.