applied sciences

MDPI

Article

Evaluating the Usability and Functionality of Intelligent Source Code Completion Assistants: A Comprehensive Review

Tilen Hliš *, Luka Četina, Tina Beranič and Luka Pavlič

Faculty of Electrical Engineering and Computer Science, University of Maribor, 2000 Maribor, Slovenia; luka.cetina@um.si (L.Č.); tina.beranic@um.si (T.B.); luka.pavlic@um.si (L.P.)
* Correspondence: tilen.hlis@um.si

Abstract: As artificial intelligence advances, source code completion assistants are becoming more advanced and powerful. Existing traditional assistants are no longer up to all the developers' challenges. Traditional assistants usually present proposals in alphabetically sorted lists, which does not make a developer's tasks any easier (i.e., they still have to search and filter an appropriate proposal manually). As a possible solution to the presented issue, intelligent assistants that can classify suggestions according to relevance in particular contexts have emerged. Artificial intelligence methods have proven to be successful in solving such problems. Advanced intelligent assistants not only take into account the context of a particular source code but also, more importantly, examine other available projects in detail to extract possible patterns related to particular source code intentions. This is how intelligent assistants try to provide developers with relevant suggestions. By conducting a systematic literature review, we examined the current intelligent assistant landscape. Based on our review, we tested four intelligent assistants and compared them according to their functionality. GitHub Copilot, which stood out, allows suggestions in the form of complete source code sections. One would expect that intelligent assistants, with their outstanding functionalities, would be one of the most popular helpers in a developer's toolbox. However, through a survey we conducted among practitioners, the results, surprisingly, contradicted this idea. Although intelligent assistants promise high usability, our questionnaires indicate that usability improvements are still needed. However, our research data show that experienced developers value intelligent assistants highly, highlighting their significant utility for the experienced developers group when compared to less experienced individuals. The unexpectedly low net promoter score (NPS) for intelligent code assistants in our study was quite surprising, highlighting a stark contrast between the anticipated impact of these advanced tools and their actual reception among developers.

Keywords: intelligent assistants; source code completion; source code

Citation: Hliš, T.; Četina, L.; Beranič, T.; Pavlič, L. Evaluating the Usability and Functionality of Intelligent Source Code Completion Assistants: A Comprehensive Review. *Appl. Sci.* **2023**, *13*, 13061. https://doi.org/10.3390/app132413061

Academic Editor: Paolino Di Felice

Received: 26 October 2023
Revised: 28 November 2023
Accepted: 5 December 2023
Published: 7 December 2023

1. Introduction

With the ever-faster development of artificial intelligence, attempts are being made to introduce this technology into various professional fields. For example, the application of artificial intelligence methods has already shown results during requirements generation and processing, project planning, and intelligent software design, as well as the areas of architecture, development, testing, and analysis, among others. In the software engineering domain, we see the two main possibilities for artificial intelligence applications: (a) a natural language interpreter and (b) a tool to improve a developer's productivity by predicting and completing a source code automatically.

In order to survive in the highly competitive software development market, developers must deliver good products quickly. Many approaches and tools help developers reduce development time while improving the quality of the final product simultaneously. Among these approaches are assistants for completing a source code, which, with the inclusion of artificial intelligence methods, are on the rise again. They help developers by improving

their productivity, from reducing typing errors and common defects to suggesting entire source code segments. Even traditional code completion assistants are rich in functionality. They typically display relevant documentation in pop-up windows, provide a preview of accessible methods and object attributes, provide variable and method name completion, and enable the generation of template-based source code sections (e.g., try-catch blocks, for-each loops, etc.). However, traditional assistants cannot generate "smart" suggestions. When generating source code suggestions, they usually rely on the information about the type of the current variable and the variables that the user has already defined in the program [1]. Although they consider the already-written program, they cannot understand the developer's intentions and suggest all syntactically appropriate methods or variables [1,2].

Due to the presented limitations, intelligent source code completion assistants, which expand the scope of functionality with the help of artificial intelligence methods, are a promising alternative. Depending on the context, they can predict the developers' intent and, thus, find the most suitable methods, even adapting them to the target situation and placing them at the top of the suggestions list. They can also generate more relevant sections of source code by considering the context of the program and developers' intent (e.g., suppose that a developer creates a variable with a name that implies the use of dates. In that case, the intelligent assistant will automatically suggest and prepare a relevant section of source code that assigns a new object of the type "Date" to the variable) [3]. Although intelligent assistants are on the rise [4], only some are available to the general public in a limited range; others offer a limited set of functionalities [5]. Many intelligent assistants promise to speed up development and reduce the number of typos and defects in the source code with more relevant suggestions. Likewise, their providers claim that they cannot only complete the current sentence but also generate entire sections of relevant source code automatically. This raises the question of whether helpers are already at the stage where they benefit developers by reducing the number of defects and shortening the time of writing code.

Many intelligent assistants promise accelerated source code writing [5,6] and fewer defects [7], but the claims are not always valid in practice. Most assistants are tested in simulated rather than real development environments, so the promised and actual results are often incomparable [8]. Aye et al. [9] conducted a study in which they monitored the use of the assistants at Facebook and found that the accuracy of suggestions dropped from 46% to 20% when the assistants were tested in an actual environment. Similar conclusions were reached by Helledoorn et al. [8], who claim that synthetic performance benchmarks of assistant performance often do not provide realistic evaluations. As part of the research, they analyzed 15,000 source code addition instances performed by 66 developers. The results were compared with those from synthetic benchmark tests and were significantly different. Synthetic tests treat every word in the program as a possible candidate for completion, but this was shown to not be adequate, as developers complete certain words more often than others and some rarely. This alone can change the assistant's accuracy significantly. The source code, while testing the intelligent assistants, can also be problematic, as a local development process can be performed in a different order and context. This is another reason why evaluating intelligent assistants in a real context is not only recommended but necessary [8].

This paper aims to include a review of the current state and trends in the field of artificial intelligence-based source code completion assistants. In the paper, we present the concept of intelligent assistants, their operations, main functionalities, and artificial intelligence-based improvements. The second aim of our research was to evaluate usability, developer experience, and the probability of recommendation of the intelligent assistants. In order to achieve this, we gathered empirical data by employing an in-depth survey. Our research follows the research questions that we address in this paper (please see Figure 1):

- **RQ1:** What is the current state-of-the-art in the field of intelligent source code completion assistants?

- **RQ2**: Which intelligent source code completion assistants are currently available, and what functionalities do they offer?
- **RQ3**: How do developers rate the usefulness of intelligent source code completion assistants?

The rest of the paper is structured as follows. Section 2 describes the research method we employed to address the presented research questions. In Section 3, we present the related work, offering a comprehensive overview of prior studies on intelligent source code completion assistants and their influence on software development. The results of the systematic literature review addressing **RQ1** and **RQ2** can be found in Section 4. Section 5 showcases the empirical results (**RQ3**) from the survey. The paper concludes with a discussion, where we address our research questions and offer our final remarks.

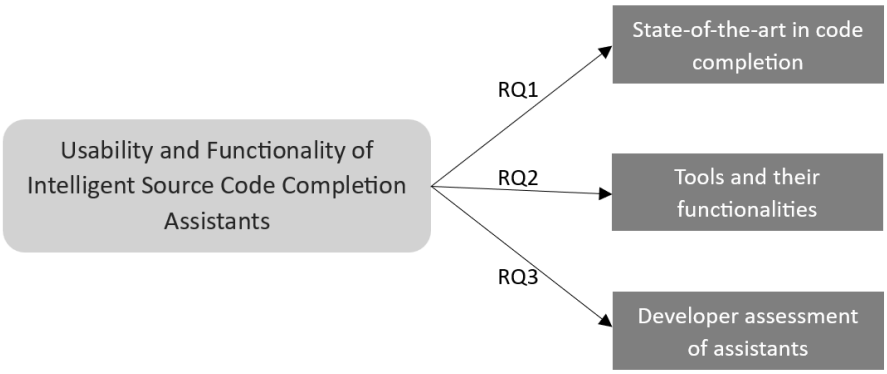

Figure 1. Research relationships.

2. Research Method

The empirical research method used to answer the research questions consisted of three steps. The first step, a systematic literature review (SLR), is described later in Section 2. The second step was a survey. We briefly describe the results of the survey in Section 5. The last step of our research method was empirical data interpretation. After using the mentioned methods, we performed the interpretation of the obtained results.

(1) **The systematic literature review (SLR) on intelligent assistants:** Our research began with an SLR of intelligent assistants, focusing on their presence within integrated development environments (IDEs) and their role in source code completion. While code completion assistants have been in use for a while, our aim was to provide a comprehensive overview of those that employ machine learning methods. This SLR was designed to investigate the available intelligent assistants and the functionalities they offer. We adhered to the systematic review methodology as outlined by Kitchenham and Charters [10], ensuring a rigorous and comprehensive approach to our research. With the rise of more advanced intelligent assistants, we delved deeper into the current state of the field. We focused primarily on the literature that deals with source code completion while writing and using machine learning methods. We excluded studies related to automatic source code generation without user intervention, abbreviations- and template-based source code generation, and those assessing existing source code with subsequent correction suggestions. This SLR, conducted in late 2022, aimed to answer research questions **RQ1** and **RQ2**. Our review utilised six major digital libraries:

- **ScienceDirect** (http://www.sciencedirect.com, accessed on 1 November 2022);
- **IEEE eXplore** (https://ieeexplore.ieee.org, accessed on 1 November 2022);
- **Scopus** (https://www.scopus.com, accessed on 1 November 2022);
- **ACM Digital Library** (https://dl.acm.org, accessed on 1 November 2022);
- **SpringerLink** (http://link.springer.com, accessed on 1 November 2022);
- **arXiv** (https://arxiv.org, accessed on 1 November 2022).

The primary objectives of the SLR were to identify the relevant literature on (a) available intelligent assistants, (b) their major functionalities, and (c) how these functionalities differed from traditional assistants.

In preparation for the SLR,and to form the relevant search strings, we conducted an **initial domain scan** by reviewing the pertinent literature from leading digital databases in software engineering. A significant observation was that the majority of recent publications focus primarily on the broader domain of source code generation. Our research, however, is centred on the application of methodologies that assist developers during the coding process. The rapid expansion of this field was evident in our preliminary review, indicating that existing studies can become outdated quickly. Consequently, our search strings were crafted to address the various perspectives of the review. As is consistent with the methodology of Kitchenham and Charters [10], the standardised search strings originate from research questions, and the process of creating keywords was systematic, involving preliminary reviews, the result of which are the search strings. The detailed findings from this SLR are presented in Section 4.

For the SLR, we formed a search string based on two groups of keywords. Group 1 included the keywords "code completion" (variation: "completions"), "code suggestion" (variant: "suggestions"), and "coding assistant" (variation: "assistance"). Group 2 included "neural", "convolutional" (variant: "convoluted"), machine learning (variant: "ml"), "bayes" (variant: "bayesian"), "intelligent" (variant: "intelligence"), "ai" (variant: "artificial intelligence"), "ai-assisted" (variant: "ai assisted"), and "ai-driven" (variant: "ai driven"). Based on the keywords, a single **aggregated search string** used to perform the SLR was as follows:

("code complet" OR "code suggest*" OR "coding assistan*") AND (neural OR ai OR "artificial intelligence" OR convolut* OR "machine learning" OR bayes* OR intelligen* OR "ai-assisted" OR "ai-driven")*

During the execution of the SLR, we also followed several **inclusion** and **exclusion criteria**. The literature inclusion criteria were as follows:

- It is in the English language;
- The full text is available in a digital database;
- It is a conference paper, journal paper, or doctoral thesis;
- It is related to the supplementing or suggesting of the source code during programming and enabling techniques and tools;
- Is related to artificial intelligence methods;
- Was published in the last 4 years (2019 and later);
- Consists solely of peer-reviewed works.

In the first search phase, this simply meant outdated, non-English, or non-accessible literature. The results of the SLR were as follows (please see Figure 2):

- **Phase 1: Initial search.** In the first phase, we performed an initial search. It returned 959 results. During our preliminary research, we discovered works [3,11] that addressed the same domain and were published between 2019 and 2020. In order to avoid redundancy and repetition, we decided to focus on works published only in the last 4 years. Consequently, we added the rule that only those pieces of literature that were not older than four years were taken into further investigation. After eliminating the duplicates, we arrived at a total of **355** items after the first phase.
- **Phase 2: Title-based screening.** In this phase, the titles of the studies were screened independently by three researchers. The results from each researcher were then combined to ensure a comprehensive and unbiased selection. This process resulted in **62** studies appropriate for inclusion in the next phase of the review.
- **Phase 3: Abstract-based screening.** Similarly, the abstracts and keywords of the studies were reviewed independently by the same three researchers. After

combining the results from each researcher, **36** studies were deemed appropriate for the next phase.

- **Phase 4: Full-text review.** The full-text review was the next phase of the SLR. Directly, it provided a set of **25** primary studies.
- **Phase 5: Snow-balling.** Finally, in the last phase, additional studies were included by employing a related work review. This is how we finished the review with **28** primary studies in the field.

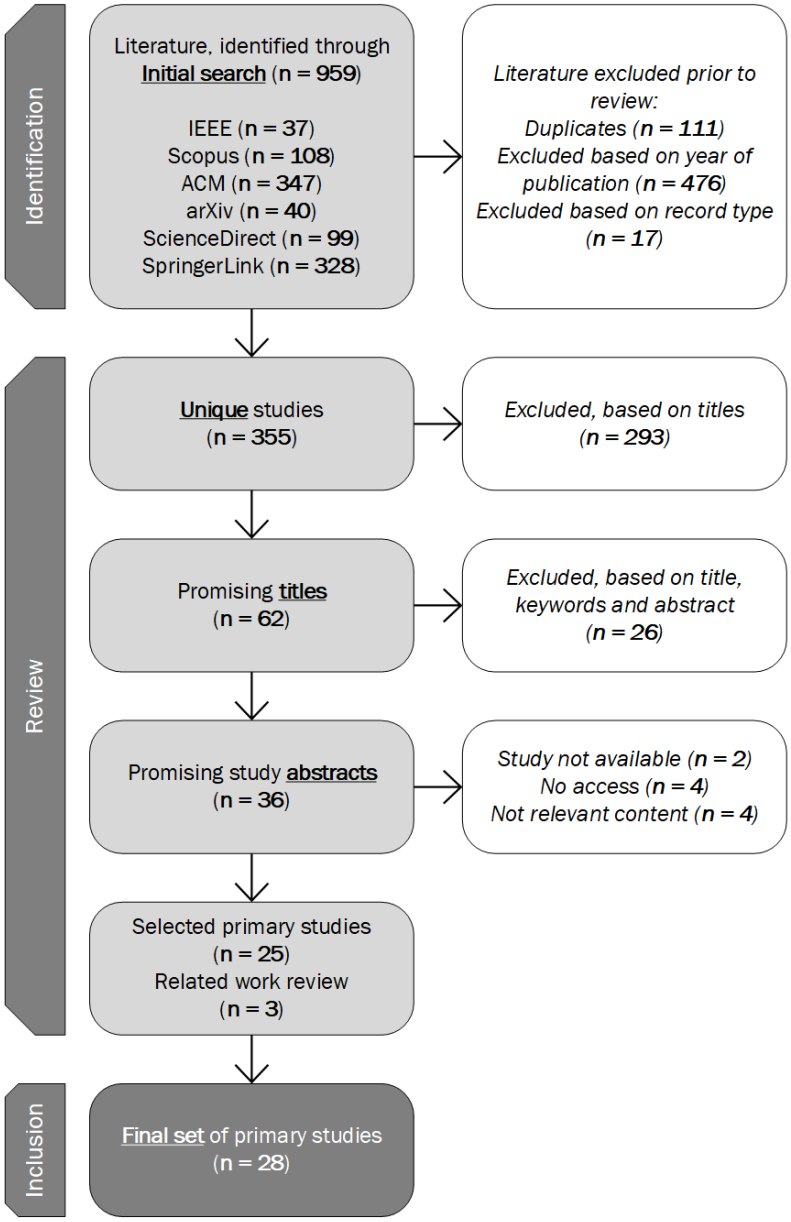

Figure 2. SLR Progression.

(2) **The Survey** was the next step, aimed at gathering the developers' opinions of the usefulness of intelligent assistants. More precisely, the survey consisted of four parts. It was designed to gain insight into the developers' backgrounds and experience with information systems (ISs) development, as well as their opinions regarding the usability of intelligent assistants, their user experience with these assistants, and the likelihood of recommending intelligent assistants to others. When designing the survey, we took into account the use of established types of surveys for different areas of research. The survey was constructed as follows:

(a) **Background and Experience:**
- Participants' experience with programming and IS development.
- Their background in using specific programming languages and development environments.

(b) **Familiarity with Intelligent Assistants:**
- Participants' familiarity with the various intelligent assistants identified from our systematic literature review.
- An option for participants to mention any assistant not listed.
- Frequency of participants' use of an intelligent assistant with the following options:
  * I have tried it;
  * I use it occasionally;
  * I use it regularly.

(c) **Usability Measurement:**
- Utilised the system usability scale (SUS) [12], comprising 10 questions.
- The scoring system ranged from 0 to 100, with higher scores indicating better usability.

(d) **User Experience Measurement:**
- Employed the short version of the user experience questionnaire (UEQ-S) [13], with eight standardised questions.
- The results measured both the pragmatic and hedonic aspects of user experience.

(e) **Recommendation Likelihood:**
- Uses the net promoter score (NPS) [14] to gauge the likelihood of participants recommending the intelligent assistant.
- The scoring system was based on a 10-point scale.

(f) **Future Use:**
- A concluding question on whether participants would continue using such intelligent assistants in the future.

(3) **The empirical data interpretation** involved various statistical methods and tools. The data from all questionnaires were combined into one file. Textual variables were converted into numerical values using the SPSS program [15], ranging from 0 to 4. The final SUS value was calculated by summing the item values and multiplying them by 2.5. The UEQ questionnaire results were computed using the provided tool [16]. Group formation was based on experience, programming language, framework, and intelligent assistants, using averages or dividing them into "I do not know at all" and "I have already heard" responses. Descriptive statistics, such as mean, standard deviation, minimum, and maximum, were calculated for each variable. The Cronbach alpha coefficient was used to assess the scale's internal reliability. Group comparisons were conducted using independent samples *t*-tests after confirming normality using the Shapiro–Wilk test. Cohen's d effect size was calculated with the Psychometric program [17]. An effect value between 0 and 0.1 was deemed insignificant, 0.2 to 0.4 as small, 0.5 to 0.7 as medium, and 0.8 to 1.0 as large.

3. Related Work

The rise of intelligent source code completion assistants corresponds with the increasing complexity of software development. These tools enhance code-writing efficiency by predicting and suggesting context-based code snippets. Their influence on software development, encompassing productivity, code quality, and user satisfaction, is under active research. This paper consolidates the existing knowledge, offering a thorough analysis of these assistants' impact. Through an extensive review, we highlight their pros and cons

and pinpoint areas for future exploration. The software development realm is increasingly exploring machine learning for code generation and retrieval, potentially revolutionising developer approaches to program logic and procedures.

The authors [18] investigated the impact of machine learning methods for code generation and retrieval on developer productivity and accuracy in the PyCharm IDE. A plugin was developed to facilitate the study, and the developers were asked to complete 14 Python programming tasks with and without the help of the plugin. The results from the qualitative surveys showed positive experiences, but the quantitative results showed inconclusive evidence of increased productivity or code quality. Further analysis identified areas for improvement and when developers preferred code generation or retrieval. The study is the first comprehensive investigation of the topic, and all data and software were released to advance the field further [18].

The next related work [19] focused on the connection between the productivity benefit of GitHub Copilot, a neural code completion system, and usage measurements from developer activity. The authors collected measurements of the acceptance rate and persistence of completions and found that the acceptance rate correlated better with reported productivity. They suggest that code suggestions in GitHub Copilot should be viewed as a conversation with a chatbot, and they plan to explore this analogy further in future work [19].

A user study [20] with 24 participants was conducted on the usability of GitHub Copilot. The study investigated users' perceptions, interaction patterns, and coping strategies when the generated code was incorrect. The results showed that although Copilot was preferred by participants for its ability to provide a starting point for programming tasks, it did not necessarily reduce task completion time or increase the success rate. The study also highlighted potential improvements for Copilot, such as better support for understanding and validating the generated code and exploring multiple solutions [20].

However, prior research has focused primarily on evaluating retrieval accuracy and the overlap of generated code with developer-written code. The actual effect of these methods on the developer workflow remains largely unexamined. While previous studies have delved into the technical aspects and immediate impacts of intelligent source code completion assistants, our article takes a holistic approach by examining the broader landscape of these tools through a systematic literature review. Furthermore, our research emphasises the unique disparity in the perceived value of these assistants between experienced and less experienced developers, shedding light on the nuanced relationship between developer experience and the utility of intelligent assistants.

4. Intelligent Source Code Completion Assistants

In the evolving landscape of software engineering, intelligent source code completion assistants have emerged as transformative tools. Unlike traditional code completers, these AI-driven systems offer context-aware code suggestions, enhancing the efficiency of the software development process. This section delves into the intricacies of these advanced tools, highlighting their mechanisms and impact on programming. In the evolving landscape of intelligent source code completion, it is vital to recognise the diversity of source code environments. These environments can be broadly classified based on their AI integration approach, with some utilising rule-based systems and others employing machine learning algorithms for more sophisticated, context-aware code suggestions. Additionally, these environments vary in their support for programming languages, ranging from language-specific to multi-language platforms. The functionality also differs, from basic code completion to advanced systems capable of generating substantial code blocks. This diversity highlights the adaptability and potential of intelligent IDE plugins across various development scenarios.

4.1. Intelligent vs. Traditional Source Code Completion Assistants

Traditional source code completion assistants usually list all the attributes or methods that are available at a certain point of the source code, usually after "." is pressed. The

developer can then select an appropriate method from an alphabetically ordered list. The process is often slower than writing the method's name manually [2,21]. As a result, the authors identified a need for more intelligent assistants that would not arrange suggestions on an alphabetical basis but rather in a relevance-based order. Artificial intelligence methods have been employed to supplement the source code, proving to be very promising in source code modelling [1,7]. The main functionalities of traditional assistants that were reported in primary studies are summarised in Table 1.

Table 1. The main functionalities of traditional source code completion assistants.

Functionalities	Sources
Completing the current word	[22–25]
Predicting the most likely next unit of source code (showing a list of suggestions)	[11,21,22,24,26–28]
Display of all possible candidates and documentation	[29–31]
Source code completion based on templates (for/while loop, iterator)	[23,30]

Unlike traditional source code completion assistants, intelligent assistants consider the context from both the current program and various other projects to recognise common patterns. By discerning these patterns, they can gauge the developer's intent. This determination often hinges on variable names or method sequences, leading to contextually relevant suggestions. Instead of offering all possible suggestions like traditional assistants, intelligent ones provide those aligned with recognised patterns, streamlining the developer's task. While the current state-of-the-art does not alleviate developers entirely, it can automate the writing of frequently used and proven code sections. This automation lets developers concentrate on more complex, creative challenges [26]. The effectiveness of this approach is contingent on vast datasets that the intelligent assistant learns from. Without the plethora of open-source projects on repositories like GitHub, AI-based code completion would not be feasible. In Table 2, we outline the functionalities that intelligent assistants offer beyond the traditional ones. We identified six in total, with *contextual program continuation* being mentioned the most frequently.

Table 2. The main functionalities of intelligent source code completion assistants.

Functionalities	Sources
Completing the current word (names of methods, variables, attributes, ...)	[21,22,30]
Generating context-sensitive program continuation suggestions	[2–4,9,22,26,28,32–35]
Displaying information about the methods and attributes of the current object (explore API)	[2,21,24]
Generating new variable or method name suggestions	[1]
Generating natural language based on source code continuation suggestions (considers method and variable names and comments)	[7]

The primary function of intelligent assistants is their ability to determine which elements come after the current one based on the existing source code [25]. The intelligent assistants try to relieve developers from thinking about the names of methods, variables, and other source code elements without requiring developers to write a single letter. When developers are under constant pressure to deliver high-quality source code in the shortest possible time, using tools that allow the generation of program continuation suggestions is essential. The tools can save a lot of work [34], speed up development [7], and, as a result, increase the productivity of developers [29,30]. Assistants based on artificial intelligence differ from traditional ones in that, when they generate suggestions for program continuation, they use the information available not only at the time of compilation but also from common patterns found in various freely accessible repositories [21]. Thus, in addition to the already defined source code elements, they can propose those not yet present in the local context (eng. zero-day source code tokens) [1,22]. An essential feature of intelligent assistants is that they generate suggestions and rank them by relevance. The

best suggestion should always appear at the top of the list so that developers only need to check the first few entries instead of searching through the entire list. Through machine learning methods, intelligent assistants can generate longer and more complex suggestions, ranging from simple words to complete sections of source code.

The following standard functionality *suggests a suitable API and shows an API usage example (including parameters) adapted to the current context*. Developers do not write all parts of the program source code themselves but find a library that meets their requirements and then use it. Since so many libraries exist, deciding which one to use is often quite challenging. Here, again, intelligent assistants can help developers make their work easier by suggesting which library to use at a given location [23] by considering the already written program [28]. The intelligent assistants also show an example of the proposed API, which allows developers to instantly assess whether the API is suitable for them and learn how to use it [7,23,28]. Intelligent assistants maximise the relevance of the displayed use case and adapt it to the needs of the current context with the help of artificial intelligence [28]. Some intelligent assistants can use objects from the local context as parameters when generating the API use case.

Assistants based on artificial intelligence enhance traditional ones by not only suggesting the existing variables and methods defined in the program but also by offering advanced naming suggestions. Specifically, they can *predict meaningful names for variables or methods based on the context*, even proposing entirely new names that are not present in the local context [1,22]. This prediction considers the names of existing variables, methods, and even comments within the program [7]. Similar to traditional assistants, intelligent ones allow developers to *explore APIs by displaying a list of all available methods and attributes*. However, these results are typically ranked by relevance rather than in alphabetical order [32]. With the help of these data, they can determine the developer's intention and generate a proposal that is as relevant as possible to the developer. In practice, the assistant will gather enough data based on the comment and method name to write the method body. As a rule, it will not be able to do this in all cases. However, it will make the developer's work significantly easier by *generating suggestions based on natural language*, even if only occasionally.

4.2. Leading Intelligent Source Code Completion Assistant Comparison

After a systematic literature review, we identified eight intelligent assistants (Table 3) from primary studies.

Table 3. Overview of intelligent assistants.

Intelligent Assistant	Accessibility	Sources
GitHub Copilot	Available as a plugin for Visual Studio Code, JetBrains IDE, and Neovim	[4,36]
Tabnine (Codota)	Available as an IDE plugin	[4,9,25,37,38]
Kite	Available as an app and IDE plugin	[4,25]
IntelliCode	Part of the VS IDE and VS code	[2,28,29,32]
IntelliCode Compose	Internal to Microsoft	[32,37]
CACHECA	Not accessible	[7]
Pythia	Internal to Microsoft	[26,33,38]
Galois	No IDE integration	[9]

GitHub Copilot has garnered significant attention recently. Developed by GitHub in collaboration with OpenAI, this assistant offers program continuation suggestions based on the context from comments and source code. It utilises the Codex language model, an evolution of the GPT-3 model by OpenAI. Codex translates natural language into source code, learning from open repositories on GitHub [5].

Tabnine (previously Codota) employs OpenAI's GPT-2 transformer. Its primary function is to predict and suggest the subsequent unit of source code, typically the next word or line [6].

Kite, similar to Tabnine, is built on OpenAI's GPT-2 model. It suggests the next word or line of source code using local context and open-source samples. In addition to code completion, Kite displays documentation for nearby objects. To use Kite, users must install an additional application alongside the development environment plugin [39].

IntelliCode, rooted in Microsoft's IntelliSense, is designed for Visual Studio and Visual Studio Code. Trained on a GPT-2 transformer, it learned from numerous public GitHub repositories. IntelliCode suggests the next program word, considering local context, repository metadata, and official documentation. *IntelliCode* Compose, an enhancement, emphasises completing entire code lines using the GPT-C model, a GPT-2 variant. It is still under development and exclusive to Microsoft developers [37].

Three other intelligent assistants identified in our primary studies include *CACHECA*, which was designed for Eclipse and based on a modified n-gram model, focusing on the current file [40]. *Pythia* is under development, leveraging the GPT-2 model trained on select high-quality open-source projects [21]. Lastly, the open-source *Galois* assistant, which is environment-independent and built on GPT-2, can suggest multiple words but not full lines of code [41].

After a comprehensive review and description of intelligent assistants for complementing the source code, we tested those that are available for use. These are GitHub Copilot, Tabnine, Kite, and IntelliCode. The examination provided the results of their capabilities, which are compared in Table 4.

Table 4. The accessible intelligent assistants.

		GitHub Copilot	**Tabnine**	**Kite**	**IntelliCode**
Current word completion		✓	✓	✓	✓
Generating a program continuation:	Word	✓	✓	✓	✓
	Line	✓	✓	✓	
	Section of source code	✓			
API exploration			✓	✓	✓
API proposal		✓	✓	✓	✓
Naming variables		✓			
Source code generation based on natural language		✓	✓		
Learning the model on your own source code (for enterprises)		✓	✓	✓	✓

Recently, the landscape of intelligent code completion has been profoundly transformed by the advent of more sophisticated AI models. Prominently, GPT-3 and GPT-4, developed by OpenAI [42], have emerged as game-changers. These models have significantly advanced the capabilities of intelligent assistants in understanding and generating code, offering a more context-aware, nuanced approach than their predecessors [43]. GitHub Copilot, initially leveraging OpenAI's Codex model based on GPT-3, has now been updated to GPT-4 with its new version, Copilot X, introducing features like Copilot Chat [5]. This enables a ChatGPT-like experience, allowing developers to discuss specific code segments for better understanding or modification, even via voice input. Another notable tool is Codeium [44], which provides AI-generated autocomplete in over 20 programming languages and integrates directly with various IDEs. It accelerates development by offering rapid multiline code suggestions and reducing the time spent on searching APIs and documentation [44]. The emergence of such tools is not just limited to IDE plugins but extends to various platforms, including those specifically designed for developers. The development of these tools is progressing so rapidly that new and innovative solutions are being introduced in short periods, continually enhancing the programming landscape.

4.3. Issues and Main Challenges in Intelligent Source Code Completion

While intelligent source code completion tools have significantly enhanced software development, several open issues and challenges remain to be addressed. This section explores these challenges and their implications for the effectiveness of these tools.

In the field of intelligent source code completion, several challenges persist that impact the effectiveness of these advanced tools. A key area is the alignment of automated testing methodologies with the suggestions made by intelligent code completion tools. The accuracy and relevance of these suggestions are paramount, as they can significantly influence the efficiency and effectiveness of automated testing processes [45].

Furthermore, the formal verification and validation of code generated by AI assistants present unique challenges. Ensuring the reliability and correctness of this code is critical, particularly in high-stakes applications where the consequences of errors are significant [20].

Additionally, a major limitation of current AI methodologies is their ability to fully understand and predict developer intent. This limitation can compromise the quality and applicability of the code completion suggestions, underscoring the need for ongoing research to enhance the interpretative capabilities of AI in software development environments [46].

These challenges highlight the need for continued research and development in the field of intelligent source code completion. Addressing these issues will not only improve the current tools but also pave the way for more advanced and reliable AI-driven development environments.

5. The Survey Results

In order to answer RQ3, which addresses the usability of intelligent assistants as perceived by users, we designed a survey questionnaire. Developers with experience using at least one of the intelligent assistants participated in the survey, which was conducted in late 2022/early 2023. The respondents, who had diverse demographic backgrounds and experience in IS development, were invited to the survey via a group on a social network. Out of the participants, **68** individuals with experience in IS development submitted a fully completed questionnaire, and these responses were used for further data analysis.

In order to create a profile of respondents in the survey, it was necessary to ask specific questions about the characteristics of the respondents in the survey. The results are presented in Table 5 and Figure 3.

Table 5. Characteristics of respondents.

	Mean	Standard Deviation	MIN	MAX
Programming experience [Years]	8.8	7.2	2	35
Experience with the development of IS [Years]	6.3	7.5	1	30

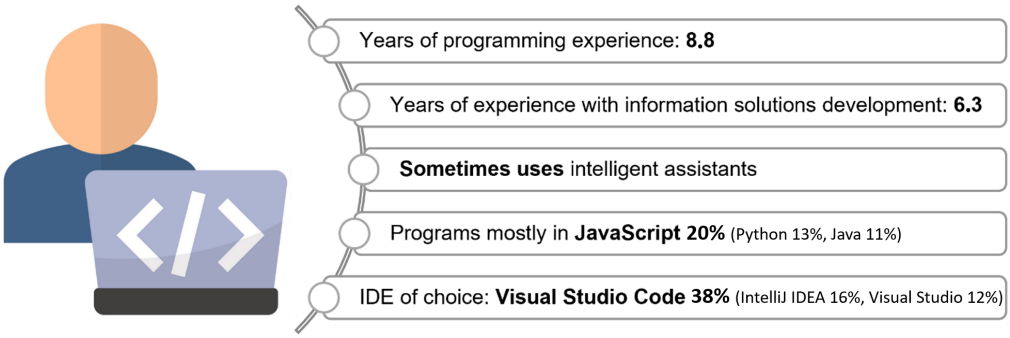

Figure 3. Average respondent.

Table 6 shows the averages, standard deviations, the lowest and highest measured values, and the results of the normality test of the variables that were measured as part of

the statistical analysis of the survey results. The table presents the NPS value for individuals, and the total NPS value is given in the paragraph below.

Table 6. Descriptive statistics and the Shapiro–Wilk test for measuring variables.

	Mean	Standard Deviation	MIN	MAX	Shapiro-Wilk
SUS	65.9	16.01	30	95	0.292
UEQ-s	0.97	1.13	−1.5	3	0.425
UEQ-p	0.84	1.14	−1.5	3	0.336
UEQ-h	1.1	1.37	−3	3	0.032
NPS	6.81	2.57	0	10	0.001
Knowledge of intelligent assistants	3	1.3	1	4	<0.001

According to the official **SUS** questionnaire scale, the participants rated the intelligent assistant's usefulness as **65.9** (grade **C**, based on the SUS rating scale), or **ok** but **close to satisfactory**. In our sample, the **internal reliability** or alpha coefficient of the scale was **0.87**.

The user experience was assessed as **average**, or **0.97**, which is also consistent with other studies and the official scale for interpreting **UEQ** questionnaires. In our sample, the **internal reliability** or alpha coefficient of the scale was **0.89**.

The average participant would suggest using an intelligent assistant to others in approximately **slightly more than two-thirds of cases**.

The total NPS value was **−11**, which means that the participants were **not satisfied** with it, and they felt that it should be improved significantly in order for the participants to recommend it to others.

However, **82%** of the participants stated that they **would continue to use the intelligent assistants**. The average participant had at least almost **9** years of experience in programming and around **6.3** years of experience in IS development. Almost **70%** of respondents used intelligent assistants at least occasionally to complete program code. It can be seen from Table 6 that the first three variables (SUS, UEQ-s, and UEQ-p) are **typically distributed**, whereas the rest are **non-normally distributed**.

Given the rapid evolution and complexity of the software development landscape, it is imperative to categorise professionals based on their specific expertise and experience. This division is not arbitrary but is rooted in the nuances of programming, the intricacies of IS development, and the specialised knowledge required for intelligent assistants. By distinguishing between the number of years of programming experience, the number of years dedicated to the development of IS, and familiarity with intelligent assistants, we aimed to provide a more granular understanding of the capabilities of the professionals who were included in the survey.

Table 7 shows the results of the participants' comparisons according to their **years of programming experience**. The **less experienced group** represents those **without more than 5 years** of programming experience. In the **more experienced group**, we have included those with **6 years** or more of programming experience [47].

Table 7 shows that, on average, the **experienced** participants rated the intelligent assistants as **significantly more helpful and rated them as good**, whereas the **less experienced** participants rated them as **poor**.

Additionally, **experienced** users reported an **above-average positive** user experience, whereas the **less experienced** users, on average, rated the user experience as entirely **average**.

At the same time, the **experienced** respondents reported being **more likely to recommend** the intelligent assistant than inexperienced participants. For the comparison of **usability evaluation** and the **pragmatic aspect** of the user experience, the **effect size** was slightly **more than medium**, as indicated by the results of the *t*-test, which showed that the differences between the groups were **statistically significant**.

Table 7. Descriptive statistics, Shapiro–Wilk test, independent samples *t*-test results, and effect sizes according to programming experience.

	Less Experienced N = 28			**Experienced N = 40**					
	Mean	Standard Deviation	Shapiro–Wilk	Mean	Standard Deviation	Shapiro–Wilk	t(51)	p	d
SUS	59.88	13.87	0.17	**69.55**	16.3	0.338	−2.21	0.032	0.64
UEQ-s	0.82	1.08	0.956	**1.07**	1.16	0.247	−0.77	0.446	0.22
UEQ-p	0.45	1.11	0.893	**1.08**	1.11	0.336	−2.02	0.049	0.57
UEQ-h	**1.19**	1.25	0.596	1.04	1.46	0.08	0.38	0.709	−0.11
NPS	6.4	2.44	0.167	7.06	2.65	0.003	−0.91	0.369	0.26

The **effect size** was **small** for the **hedonic aspect of user experience**, **overall user experience**, and the **likelihood of a recommendation**. The differences in the latter variables did not turn out to be statistically significant.

Table 8 shows the results of the comparison of the participants in the survey according to their **years of experience in the development of IS**. Again, those with a **maximum of 5 years** of experience developing IS are considered **less experienced**. In the group of **more experienced** people, we have included those with **6 years or more** of experience in developing IS [47].

Table 8. Descriptive statistics, Shapiro–Wilk test, and *t*-test results for independent samples and effect sizes according to experience in developing IS.

	Less Experienced N = 40			**Experienced N = 28**					
	Mean	Standard Deviation	Shapiro–Wilk	Mean	Standard Deviation	Shapiro–Wilk	t(51)	p	d
SUS	62.96	14.94	0.151	**70.75**	16.9	0.406	−1.75	0.086	0.49
UEQ-s	0.77	1.16	0.786	**1.32**	1.01	0.839	−1.76	0.085	0.51
UEQ-p	0.65	1.1	0.652	**1.16**	1.16	0.671	−1.61	0.115	0.45
UEQ-h	0.88	1.48	0.236	**1.45**	1.12	0.153	−1.49	0.141	0.44
NPS	6.24	2.68	0.021	7.75	2.1	0.027	−2.15	0.037	0.63

Table 8 shows that, on average, the **experienced** participants rated the intelligent assistants as **more useful**. The relationship between the groups is similar to the comparison in Table 7, where we compared participants based on programming experience.

Overall, the **usability averages** in Table 8 are **higher** in comparison with Table 7. When comparing the average **user experience** rating, it turned out that the **experienced** respondents rated it significantly **above average**, almost good. In contrast, the **less experienced** users rated the **user experience** as almost **below average**.

The **experienced** group gave a **higher average recommendation score** than the average recommendation score of all respondents; those with **less experience** were assigned a **lower average score** than the total. The difference is also indicated by the results of the *t*-test, which showed that the difference was statistically significant, and this was confirmed by the higher effect size. The remaining variable differences appear not to be statistically significant, although the effect sizes are all medium.

Table 9 shows the results of the participants' **comparisons according to their knowledge of the intelligent assistants**. The **less experienced** group includes those who answered "I have already heard" or "I have tried it" to the question about their familiarity with intelligent assistants. In the group of **experienced** participants, we included those who answered the question with "I sometimes use it", and "I use it regularly".

Table 9. Descriptive statistics, Shapiro–Wilk test, and *t*-test results for independent samples and effect sizes according to knowledge of intelligent assistants.

	Less Experienced N = 21			Experienced N = 47					
	Mean	Standard Deviation	Shapiro–Wilk	Mean	Standard Deviation	Shapiro–Wilk	t(51)	p	d
SUS	58.46	11.57	0.611	**68.31**	16.62	0.175	−1.98	0.053	0.67
UEQ-s	0.63	1.04	0.737	**1.08**	1.14	0.299	−1.27	0.21	0.41
UEQ-p	0.23	1.01	0.806	**1.04**	1.12	0.269	−2.33	0.024	0.76
UEQ-h	1.04	1.3	0.98	1.11	1.41	0.045	−0.15	0.879	0.05
NPS	6.77	2.62	0.09	6.83	2.58	0.004	−0.07	0.946	0.02

6. Discussion

In this section, we delve into the findings and implications of our study, addressing each research question (RQ) in turn. We first explore the current state-of-the-art in the field of intelligent source code completion assistants (RQ1). Subsequently, we discuss the available intelligent assistants and their functionalities (RQ2). Finally, we examine how developers rated the usefulness of these intelligent assistants (RQ3).

RQ1: *What is the current state-of-the-art in the field of intelligent source code completion assistants?*

Our systematic literature review clarified the concept and primary functions of intelligent assistants. These assistants outperform traditional ones by using machine learning to detect patterns in extensive source code databases [4]. By analysing diverse code from various sources, they recognise common patterns and understand developer intent better, offering more effective assistance.

Some intelligent assistants interpret natural language, allowing them to leverage comments and variable names in code continuation. Consequently, they offer developers more pertinent and precise suggestions [7,36].

We pinpointed the key functionalities of intelligent assistants. The most common is the **generation of program continuation suggestions based on context** [25,38], aligning with the essence of source code completion.

Assistants provide developers primarily with program continuation suggestions, reducing development time [7,29]. In addition to faster development, they promise quicker API searches [23] and enhanced code quality [7]. The essential functionalities include word completion [21], API exploration [2], and suggestion [28]. However, automatic variable naming [1] and natural language-based code generation [7] remain uncommon.

RQ2: *What intelligent source code completion assistants are currently available, and what functionalities do they offer?*

Many authors develop source code-completion approaches, but few concretise them and use them to create one of the intelligent assistants. When we set about testing and describing the intelligent assistants mentioned in Table 3, we realised that some were still in the testing phase or unavailable to the general public, so we could not test them. Therefore, we included four intelligent assistants in the comparison.

GitHub Copilot turned out to be the **most advanced**, as it supports even the most demanding functionalities, such as generating entire sections of source code, naming variables, and generating source code based on natural language. However, it does not support some basic functionalities, such as displaying all the accessible attributes and methods of an object.

Regarding capabilities, it is followed by **Tabnine**, which is inferior in that it can suggest an entire line but not name variables. However, it allows API exploration and is available for most IDEs.

Kite is similar in capability to Tabnine, except that it does not support natural language source code generation.

IntelliCode was the least capable, generating only one-word suggestions.

RQ3: *How do developers rate the usefulness of intelligent source code completion assistants?*

Following the findings in [8], our survey revealed insights into the use of intelligent assistants. The main conclusions are the following:

The intelligent assistants' **usefulness** was rated as **OK**, with the **user experience** being **average**. Interestingly, the **hedonic aspect** (pleasure of use) was rated higher than the **pragmatic** one, suggesting the assistant is enjoyable to use but not always effective.

From Table 9, those participants familiar with the assistants found them more beneficial, rating both the pragmatic aspect and overall user experience higher. In contrast, **non-experts** gave neutral to modest ratings, especially for usability.

A significant factor affecting the ratings was the relevance of the assistant's suggestions. The **NPS** value was $-$**11**, indicating dissatisfaction. However, **82%** would continue using AI-based assistants, hinting at their potential.

Experienced developers found the assistant more practical, appreciating its hedonic qualities. They likely discerned relevant suggestions better and valued non-functional aspects like ease of use. In contrast, less experienced users, relying more on the assistant, might have been misled by irrelevant suggestions.

When comparing ratings based on prior knowledge of AI assistants, there were notable differences. Those familiar with such technology rated it positively, whereas those unfamiliar with it were more critical.

In summary, while intelligent assistants are seen as promising, their effectiveness, especially in suggestion relevance, needs improvement. Their appeal seems higher among experienced developers and those familiar with AI-based tools.

7. Limitations and Threats to Validity

We recognise some limitations and threats to the validity of the research presented in this paper and its outcomes, and we advise readers to consider these aspects when interpreting our findings.

Our research began by inviting participants to take the survey through a social network group, targeting candidates with experience in software development. However, there exists a possibility that not all respondents genuinely have experience with software development using intelligent assistants, even though we inquired about their experience in this domain.

A notable limitation is the availability of many of the intelligent assistants. Many of those identified through our systematic literature review are still under development and not widely accessible. As a result, our comparison was limited to specific tools, namely, GitHub Copilot, Tabnine, Kite, and IntelliCode.

The domain of our research is evolving rapidly, especially with tools like GitHub Copilot introducing advanced functionalities. This rapid evolution might lead to changes in the landscape of intelligent assistants, which could impact the relevance of our findings over time.

While we categorised respondents based on their years of programming experience, it is essential to highlight that the validity of our statistical analysis might be influenced by how participants self-assessed their expertise. Their perception of their own experience might differ from the actual years of experience they possess.

In our survey, we focused on structured questions and did not include free-text responses. While this approach streamlined the data analysis, it limited our ability to gather in-depth qualitative feedback from participants about their experiences with intelligent code assistants.

Lastly, given the agile nature of our research domain, there is a potential threat to the credibility of our findings. The research stages spanned an extended period, during which the field might have undergone changes, affecting the current relevance of our findings.

8. Conclusions

AI-based or intelligent source code completion assistants have the potential to improve developer productivity hugely through their ability to assimilate contextual information about the developer's current activity and then generate substantial completions in response. In this paper, we defined the concept of intelligent assistants and identified their main functionalities. We found that the critical functionality of intelligent assistants is the "*context-based generation of program continuation suggestions*", as 13 authors have highlighted in their publications. We listed and presented the intelligent assistants mentioned by the authors in the reviewed literature and compared their functionalities. We found that many of the intelligent assistants are still in development and are not widely available; therefore, we only evaluated *GitHub Copilot*, *Tabnine*, *Kite*, and *IntelliCode*. The comparison confirmed the conclusion of the reviewed literature: completing entire sections of source code is very demanding, and only *GitHub Copilot* can do this. Similarly, the functionality of naming variables and generating source code based on natural language turned out to be challenging since only *GitHub Copilot* could do the first, and only *Tabnine* could do the second. Based on the data obtained from the statistical analysis of the survey questionnaire results, we conclude that, from the point of view of the usefulness of intelligent assistants, much more needs to be done for such assistants to be accepted as very useful. This may be due to the fact that there is currently very little or, practically, only one intelligent assistant available (*GitHub Copilot*) that shows truly out-of-the-ordinary results in performing its primary task—that is, predicting and supplementing the source code according to the user's needs and desires. Perhaps with the development of more and more of such assistants, the general opinion about their usefulness will change. Developers will also have to pay special attention to the simplicity and pragmatic aspects of the user experience of intelligent assistants, which has proven to be exceptionally poorly received, especially among inexperienced developers.

In conclusion, our study sheds light on the current state-of-the-art in intelligent source code completion assistants and provides valuable insights into their functionalities and user perceptions. While our findings indicate that intelligent assistants do not enhance the quality of existing code significantly or generally reduce the number of coding errors, as is expected, they do show promise in expediting the code-writing process. Surprisingly, our results also revealed that the pragmatic aspect of the user experience received lower ratings compared to the hedonic aspect, suggesting a need for more practical and goal-oriented assistance. Given the promising capabilities of intelligent assistants like GitHub Copilot, it becomes imperative for future research to delve deeper into their real-world impact. A comprehensive technical evaluation of Copilot, focusing on its influence on the programming process and the resultant source code quality, would provide invaluable insights. Such an experiment would not only gauge its practical utility but also set a benchmark for the evolution of intelligent assistants in the software development domain. Moreover, it is noteworthy that the emergence of advanced services like ChatGPT and BARD, with their advanced natural language processing capabilities, may potentially replace existing intelligent assistants in the future. These services have the potential to offer more comprehensive and context-aware assistance, surpassing the limitations of traditional and intelligent assistants. As the field continues to evolve, it is crucial to address the identified limitations and work towards enhancing the relevance, usability, and overall effectiveness of intelligent assistants.

Furthermore, in planning future work, we intend to conduct a thorough examination of the impact of these assistants on the speed and quality of code development, utilising static metrics. Although a preliminary study indicates no significant influence, this will undoubtedly be a subject of further research, as understanding the technical capabilities and limitations of these tools is essential for their development and refinement.

Author Contributions: Conceptualization, L.Č.; Investigation, L.Č., T.B. and L.P.; Methodology, T.H., T.B. and L.P.; Resources, T.H., T.B. and L.P.; Supervision, T.B. and L.P.; Validation, T.H., L.Č., T.B. and L.P.; Visualization, T.H., L.Č., T.B. and L.P.; Writing—original draft, T.H., L.Č., T.B. and L.P.; Writing—review & editing, T.H., L.Č., T.B. and L.P. All authors have read and agreed to the published version of the manuscript.

Funding: The project was co-financed by the Republic of Slovenia, Ministry of Higher Education, Science and Innovation, and the European Union—NextGenerationEU. The project was implemented in accordance with the Smart, Sustainable and Inclusive Growth development area, Strengthening of Competencies component, especially for digital competencies and those required by new professions and the green transition (C3 K5), for the investment measure Investment F. Implementation of pilot projects, the results of which will serve as a basis for the preparation of grounds for the reform of higher education for a green and resilient transition to Society 5.0: the project Pilot projects for the Reform of Higher Education for a Green and Resilient Transition.

Data Availability Statement: Data are contained within the article.

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

References

1. Hussain, Y.; Huang, Z.; Zhou, Y.; Wang, S. DeepVS: An Efficient and Generic Approach for Source Code Modeling Usage. *arXiv* **2019**, arXiv:1910.06500.
2. Svyatkovskiy, A.; Lee, S.; Hadjitofi, A.; Riechert, M.; Franco, J.; Allamanis, M. Fast and Memory-Efficient Neural Code Completion. *arXiv* **2020**, arXiv:2004.13651.
3. Yang, B.; Zhang, N.; Li, S.; Xia, X. Survey of intelligent code completion. *Ruan Jian Xue Bao/J. Softw.* **2020**, *31*, 1435. [CrossRef]
4. Ernst, N.A.; Bavota, G. AI-Driven Development Is Here: Should You Worry? *IEEE Softw.* **2022**, *39*, 106–110. [CrossRef]
5. GitHub Copilot. GitHub Copilot Your AI Pair Programmer . 2022. Available online: https://copilot.github.com/ (accessed on 24 October 2022).
6. Tabnine. Tabnine Docs. 2022. Available online: https://www.tabnine.com/ (accessed on 24 October 2022).
7. Hussain, Y.; Huang, Z.; Zhou, Y.; Wang, S. Deep Transfer Learning for Source Code Modeling. *arXiv* **2019**, arXiv:1910.05493.
8. Hellendoorn, V.J.; Proksch, S.; Gall, H.C.; Bacchelli, A. When Code Completion Fails: A Case Study on Real-World Completions. In Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 25–31 May 2019; pp. 960–970. [CrossRef]
9. Aye, G.A.; Kim, S.; Li, H. Learning Autocompletion from Real-World Datasets. *arXiv* **2020**, arXiv:2011.04542.
10. Kitchenham, B.A. Systematic Review in Software Engineering: Where We Are and Where We Should Be Going. In Proceedings of the 2nd International Workshop on Evidential Assessment of Software Technologies, EAST '12, Lund, Sweden, 19–20 September 2012; pp. 1–2. [CrossRef]
11. Hu, X.; Li, G.; Liu, F.; Jin, Z. Program Generation and Code Completion Techniques Based on Deep Learning: Literature Review. *Ruan Jian Xue Bao/J. Softw.* **2019**, *30*, 1223. [CrossRef]
12. Brooke, J. SUS: A quick and dirty usability scale. *Usability Eval. Ind.* **1995**, *189*, 189–194.
13. Schrepp, M.; Hinderks, A.; Thomaschewski, J. Design and Evaluation of a Short Version of the User Experience Questionnaire (UEQ-S). *Int. J. Interact. Multimed. Artif. Intell.* **2017**, *4*, 103. [CrossRef]
14. Reichheld, F. The One Number You Need to Grow. *Harv. Bus. Rev.* **2003**, *81*, 46–55.
15. IBM. IBM SPSS Statistics. Available online: https://www.ibm.com/products/spss-statistics (accessed on 2 December 2023).
16. UEQ-Online. Available online: https://www.ueq-online.org/ (accessed on 2 December 2023).
17. Lenhard, W.; Lenhard, A. Computation of Effect Sizes. 2012. Available online: https://www.psychometrica.de/effect_size.html (accessed on 24 October 2022).
18. Xu, F.F.; Vasilescu, B.; Neubig, G. In-IDE Code Generation from Natural Language: Promise and Challenges. *arXiv* **2021**, arXiv:2101.11149. https://doi.org/10.48550/ARXIV.2101.11149.
19. Ziegler, A.; Kalliamvakou, E.; Simister, S.; Sittampalam, G.; Li, A.; Rice, A.; Rifkin, D.; Aftandilian, E. Productivity Assessment of Neural Code Completion. *arXiv* **2022**, arXiv:2205.06537. https://doi.org/10.48550/ARXIV.2205.06537.
20. Vaithilingam, P.; Zhang, T.; Glassman, E.L. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In Proceedings of the Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems, CHI EA '22, New Orleans, LA, USA, 29 April–5 May 2022. [CrossRef]
21. Svyatkovskiy, A.; Zhao, Y.; Fu, S.; Sundaresan, N. Pythia: AI-assisted Code Completion System. *arXiv* **2019**, arXiv:1912.00742.
22. Terada, K.; Watanobe, Y. Code Completion for Programming Education based on Recurrent Neural Network. In Proceedings of the 2019 IEEE 11th International Workshop on Computational Intelligence and Applications (IWCIA), Hiroshima, Japan, 9–10 November 2019; pp. 109–114. [CrossRef]
23. Chen, C.; Peng, X.; Sun, J.; Xing, Z.; Wang, X.; Zhao, Y.; Zhang, H.; Zhao, W. Generative API usage code recommendation with parameter concretization. *Sci. China Inf. Sci.* **2019**, *62*, 192103. [CrossRef]

24. Schumacher, M.E.H.; Le, K.T.; Andrzejak, A. Improving Code Recommendations by Combining Neural and Classical Machine Learning Approaches. In Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops, ICSEW'20, Seoul, Republic of Korea, 23–29 May 2020; pp. 476–482. [CrossRef]
25. Li, J.; Huang, R.; Li, W.; Yao, K.; Tan, W. Toward Less Hidden Cost of Code Completion with Acceptance and Ranking Models. *arXiv* **2021**, arXiv:2106.13928.
26. Kalyon, M.S.; Akgul, Y.S. A Two Phase Smart Code Editor. In Proceedings of the 2021 3rd International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA), Ankara, Turkey, 11–13 June 2021; pp. 1–4. [CrossRef]
27. Karampatsis, R.M.; Babii, H.; Robbes, R.; Sutton, C.; Janes, A. Big code != big vocabulary. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, Seoul, Republic of Korea, 27 June–19 July 2020; ACM: New York, NY, USA, 2020. [CrossRef]
28. Nguyen, P.T.; Di Rocco, J.; Di Ruscio, D.; Ochoa, L.; Degueule, T.; Di Penta, M. FOCUS: A Recommender System for Mining API Function Calls and Usage Patterns. In Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 25–31 May 2019; pp. 1050–1060. [CrossRef]
29. Yang, K.; Yu, H.; Fan, G.; Yang, X.; Huang, Z. A Graph Sequence Neural Architecture for Code Completion with Semantic Structure Features. *J. Softw. Evol. Process* **2022**, *34*, e2414. [CrossRef]
30. Nguyen, S.V.; Nguyen, T.N.; Li, Y.; Wang, S. Combining Program Analysis and Statistical Language Model for Code Statement Completion. *arXiv* **2019**, arXiv:1911.07781.
31. Zhong, C.; Yang, M.; Sun, J. JavaScript Code Suggestion Based on Deep Learning. In Proceedings of the 2019 3rd International Conference on Innovation in Artificial Intelligence, ICIAI 2019, Suzhou, China, 15–18 March 2019; pp. 145–149. [CrossRef]
32. Ciniselli, M.; Cooper, N.; Pascarella, L.; Poshyvanyk, D.; Di Penta, M.; Bavota, G. An Empirical Study on the Usage of BERT Models for Code Completion. *arXiv* **2021**, arXiv:2103.07115. https://doi.org/10.48550/ARXIV.2103.07115.
33. Wang, Y.; Li, H. Code Completion by Modeling Flattened Abstract Syntax Trees as Graphs. *arXiv* **2021**, arXiv:2103.09499. https://doi.org/10.48550/ARXIV.2103.09499.
34. Arkesteijn, Y.; Saldanha, N.; Kostense, B. Code Completion using Neural Attention and Byte Pair Encoding. *arXiv* **2020**, arXiv:2004.06343.
35. Hu, X.; Men, R.; Li, G.; Jin, Z. Deep-AutoCoder: Learning to Complete Code Precisely with Induced Code Tokens. In Proceedings of the 2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC), Milwaukee, WI, USA, 15–19 July 2019; Volume 1, pp. 159–168. [CrossRef]
36. Sobania, D.; Briesch, M.; Rothlauf, F. Choose Your Programming Copilot: A Comparison of the Program Synthesis Performance of Github Copilot and Genetic Programming. In Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '22, Boston, MA, USA, 9–13 July 2022; pp. 1019–1027. [CrossRef]
37. Svyatkovskiy, A.; Deng, S.K.; Fu, S.; Sundaresan, N. IntelliCode Compose: Code Generation Using Transformer. *arXiv* **2020**, arXiv:2005.08025.
38. Schuster, R.; Song, C.; Tromer, E.; Shmatikov, V. You Autocomplete Me: Poisoning Vulnerabilities in Neural Code Completion. *arXiv* **2020**, arXiv:2007.02220.
39. Kite. Kite-Free AI Coding Assitant and Code Auto-Complete Plugin. 2022. Available online: https://www.kite.com/ (accessed on 24 October 2022).
40. Franks, C.; Tu, Z.; Devanbu, P.; Hellendoorn, V. CACHECA: A Cache Language Model Based Code Suggestion Tool. In Proceedings of the 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Florence, Italy, 16–24 May 2015; Volume 2, pp. 705–708. [CrossRef]
41. Galois Autocompleter. 2022. Available online: https://github.com/MicrosoftDocs/intellicode/ (accessed on 24 October 2022).
42. OpenAI. Better Language Models and Their Implications. 2019. Available online: https://openai.com/blog/better-language-models/ (accessed on 24 October 2022).
43. OpenAI. GPT-4 Technical Report. *arXiv* **2023**, arXiv:2303.08774. https://doi.org/10.48550/ARXIV.2303.08774.
44. codeium. 2022. Available online: https://codeium.com/ (accessed on 24 November 2023).
45. Ricca, F.; Marchetto, A.; Stocco, A. AI-based Test Automation: A Grey Literature Analysis. In Proceedings of the 2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Valencia, Spain, 12–16 April 2021; pp. 263–270. [CrossRef]
46. Barke, S.; James, M.B.; Polikarpova, N. Grounded Copilot: How Programmers Interact with Code-Generating Models. *Proc. ACM Program. Lang.* **2023**, *7*, 85–111. [CrossRef]
47. Downey, J. *Career Paths for Programmers: Skills in Senior Software Roles*; Cambridge Scholars Publisher: Newcastle upon Tyne, UK, 2008.