In-IDE code generation models - a literature review

Varzaru Rebeca[1]

Supervisor(s): Fenia Aivaloglou[1]**, Xiaoling Zhang**[1]

[1]EEMCS, Delft University of Technology, The Netherlands

Abstract

Code generation is becoming one of the most important tools in an Integrated Development Environment (IDE) for programmers, be they novices or experts. It allows them to produce code faster, avoid typos and other mistakes, be more efficient and easily turn ideas into code. As such, there are many in-IDE code generation models to choose from, each with their own promises, functionalities and implementation techniques. This report presents the results of a literature survey into the code generation models that have been integrated into programming environments and serves as an overview and analysis of them. The results of this paper provide a set of guidelines for code generation that have been summarized from the selected literature.

1 Introduction

1.1 Background

Code generation is a broad topic that includes everything from the most simple next-token prediction to state of the art AI-driven models that are capable of solving complex problems by themselves. In the scope of this research (and in the Research Questions), code generation is used as an umbrella term, which includes both the feature of code completion and the more restricted subject of code generation as creating source code out of some sort of natural description. For clarity, in this paper we will make the distinction between *code generation* and *code completion* as separate features included under this umbrella term.
Code completion refers to the function of offering next token suggestions, often ordered alphabetically or by frequency of use, based on the already existing code. This is an essential feature that is nowadays offered by every major Integrated Development Environment (IDE) and is considered by many to be the most important one [13]. For the Eclipse IDE for example, it was found that all developers use the in-IDE code completion feature [11]. The reason behind this is that code completion offers numerous advantages, such as increasing the speed of coding and helping programmers avoid typos [2], improving productivity and helping to explore APIs [13], while also ”decreasing the typing effort and saving keystrokes” [6].
Code generation refers to taking a high-level concept and turning it into source code. For example, various code generation models [3; 21; 23] use natural language as input, which is one of the most widely used ways of representing said concepts. One of the main struggles of programmers, novices and experts alike, is knowing what needs to be done next in a program, but not knowing how to express that idea into code [21]. Thus, being able to express a concept, such as ”merge these two lists” or ”center this div” and to have the corresponding code returned, saves programmers a lot of time and effort.

1.2 Motivation

There are many approaches to code generation and numerous models available, such as GitHub Copilot or OpenAI Codex. There is also extensive research available on new, emerging approaches that aim to solve certain existing problems or improve the current standards [2; 5; 6; 16; 13; 19].
That being said, there is still no overview of all these models in one place. By filling this gap, we hope this will make it easier for future researchers to assess the existing code generation models, tools and standards. To that extent, this research analyzes all these different code generation models and proposes a summarized set of guidelines that follow from the reviewed papers. We hope this will create opportunities for future comprehensive investigations in this area, as well as serve as an overview of best practices for future development on the matter.

1.3 Research Questions

The main question that is being answered in this research is: *How have code generation models been integrated into coding environments?* In order to guide the research process, several intermediate sub-questions have been formulated:

- **RQ1:** What code generation models have been integrated into which coding environments?
- **RQ2:** What techniques have been used for these code generation models?
- **RQ3:** What indicators are used to evaluate code generation models?
- **RQ4:** What aspects should be considered when designing in-IDE code generation models?

1.4 Structure

The rest of this paper is organized in the following way. Section 2 discusses the methodology, including 2.1 Identifying relevant work and assessing the quality of studies, and 2.2 Summarizing the evidence and interpreting the findings. Section 3 presents the results and answers each individual sub-question. Section 4 presents a discussion following the results and includes 4.1 Limitations, 4.2 Responsible Research and 4.3 Threats to validity. Finally, Section 5 deals with the conclusion.

2 Methodology

This review is set-up as a systematic literature review, which Snyder defines as a ”research method and process for identifying and critically appraising relevant research, as well as for collecting and analyzing data from said research” [15]. This method was chosen because it is a rigorous way to select the most important works that are to be included in this review. The steps taken, outlined by Khan et al. [7] are the following: (1) formulating the research question, (2) identifying relevant work, (3) assessing the quality of studies, (4) summarizing the evidence and (5) interpreting the findings. Step (1) was outlined in Section 1.3.

Code generation models	Coding environments
code generat*	cod* environment
code model*	integrated development environment
generate code	IDE
code generat* model*	web programming environment
	programming environment

Table 1: Search table

2.1 Identifying relevant work and assessing the quality of studies

The relevant literature has been retrieved using the following databases:

- Google Scholar
- Scopus
- Web of Science

The selection of the final papers was done in three steps:

1. Defining the search query
2. Retrieving the papers
3. Eligibility check

Defining the search query

An initial search query was formulated starting from the main research question *"How have code generation models been integrated into coding environments?"*, by identifying the main two concepts, namely code generation models and coding environments, with a secondary term being the verb "integrate". These concepts were then used to create search Table 1, from which the following search queries were developed:

1. ("code generat*" OR "code model*" OR "generate code" OR "code generat* model*") AND ("cod* environment" OR "integrated development environment" OR "IDE" OR "web programming environment" OR "programming environment")
2. ("code generat*" OR "code model*" OR "generate code" OR "code generat* model*") AND ("cod* environment" OR "integrated development environment" OR "IDE" OR "web programming environment" OR "programming environment") AND "integrat*"

Search query 1 was used on each of the aforementioned databases, followed by search 2 to identify any papers that might have been missed, and the first page of results was considered for each of them (search date 28/04/2023), subject to the following exclusion criteria:

- The paper is not written in English
- The paper is a duplicate or an older version of another selected paper.

By reading the abstracts of the selected papers and analyzing their usefulness in answering this specific research question, the results were deemed insufficient, so the papers were used to find other relevant terms to use in the search query for more accurate results. After becoming more acquainted with the existing research, as well as terminology, the following updated search query was created:

- ("Large Language Models" OR "code generat*" OR "LLM" OR "code completion") AND ("Coding Environment" OR "ide" OR "Integrated Development Environment" OR "programming environment")

Retrieving the papers

This search query was used on the platforms mentioned in the previous section (search date 11/05/2023). The results were restricted to the past 5 years, but due to the time constraints of the project, not all results could be included. As such, only the first page of results was retrieved from each one. After applying the same exclusion criteria as before, the 42 remaining papers were organised using a reference management software.

It was noticed that none of them were published in the current year (2023), so the bias of automatically sorting results by relevance was considered. After performing the same search, but sorting the results by the most recently published, several highly relevant papers were discovered that did not appear on the first page of results before. So, to account for this relevance bias and ensure that the most recent developments are considered for this paper, we decided to add an extra round of searching with the results restricted to the year 2023. These papers were retrieved following the same exclusion criteria as before and then went through the eligibility check with the other 42 papers, now making a total of 54.

Eligibility check

The last step in selecting the final papers is filtering the already retrieved papers. To that extent, we will use the following inclusion and exclusion criteria:

Exclusion criteria:

1. The paper is not peer-reviewed
2. The full text of the paper is not available

Inclusion criteria:

1. Code generation must be the main focus of the paper - papers in which code generation is simply used as a tool to aid in the study of another concept will not be included
2. The subject must be approached from a computer science point of view - papers that use code generation but are focused on other fields of study will be excluded
3. The paper must consider said code generation model in the context of a programming environment - papers where code generation is used for UML for example will be discarded

The eligibility check was done in two iterations. First by reading just the abstracts of all the papers, the second by viewing the full text, each time applying the aforementioned inclusion and exclusion criteria. The remaining papers will be used in the literature review. The whole process is summarized in Figure 1.

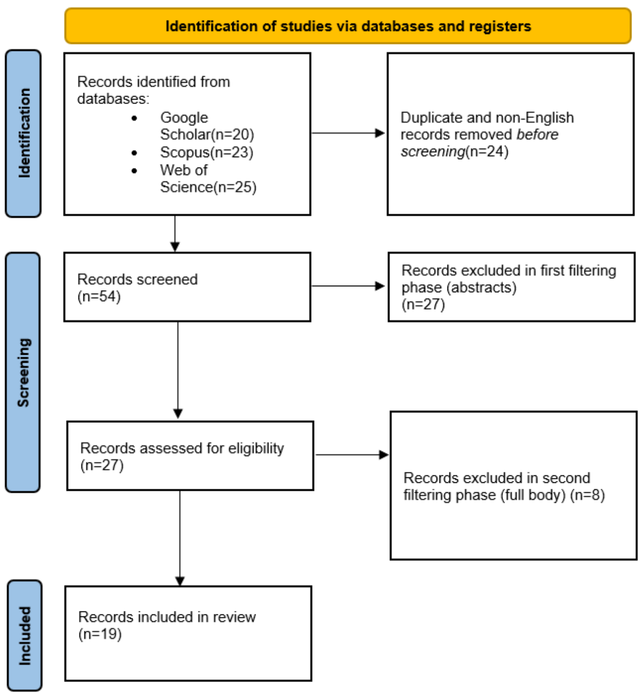
Identification of studies via databases and registers
Identification
Records identified from databases:
Google Scholar(n=20)
Scopus(n=23)
Web of Science(n=25)
Duplicate and non-English records removed *before screening*(n=24)
Screening
Records screened (n=54)
Records excluded in first filtering phase (abstracts) (n=27)
Records assessed for eligibility (n=27)
Records excluded in second filtering phase (full body) (n=8)
Included
Records included in review (n=19)

Figure 1: PRISMA flow diagram

2.2 Summarizing the evidence and interpreting the findings

The 19 eligible papers were imported into Atlas.ti, where the information extraction and data analysis were conducted. We started reading the papers, starting with the most recent ones, and applying codes in a bottom-up approach, by identifying relevant and recurring concepts as the papers were reviewed and tagging fragments of the papers accordingly.
To ensure consistency and completeness, the coding was done in two iterations. The purpose of the first iteration was to familiarize ourselves with the concepts and terminology used and compose a comprehensive list of codes, for all research sub-questions in parallel. The second one aimed to ensure that all codes were appropriately tagged in the included literature, while also serving as a chance to deepen the researcher's understanding of the concepts at hand.
More concretely, for the first sub-question all mentions of specific in-IDE code generation models were tagged and then reviewed for inclusion. For the second sub-question, we tagged all sections explaining the underlying code generation techniques, with sub-codes such as "transformer", "natural language", "deep learning", etc. to highlight explanations of the more recurring methods. In order to answer the third sub-question, we directed our attention towards the evaluation sections of the reviewed papers, tagging all indicators used by the researchers. Finally, for the fourth sub-question we highlighted all the best practices and guidelines that often accompany either the discussion section or the user feedback of an experiment.
In the end, all the information was aggregated according to the tagged concepts, and then reviewed and incorporated in the results section.

Model	IDE
IntelliCode [16]	Visual Studio Code
NL2CODE [21]	Python PyCharm
IntelliSense [16]	Visual Studio Code
Codota [16]	IntelliJ IDEA
TabNine [16]	IntelliJ IDEA, Visual Studio Code, etc.
AiXcoder [24]	IntelliJ IDEA, CLion, GoLand, PyCharm, WebStorm, Visual Studio Code, Eclipse
Kite [24]	
HISyn [23]	Visual Studio Code
OpenAI Codex [1]	
DeepMind AlphaCode [1]	
Amazon CodeWhisperer [1]	JetBrains, Visual Studio Code
GitHub Copilot [1]	JetBrains, Visual Studio Code, NeoVim

Table 2: Code generation models and IDEs

3 Results

3.1 RQ1: Code generation models and their IDEs

For conciseness, the complete results of research question 1 are presented in Table 2, with the first column containing the name of the model and the second column denoting the IDE where it has been integrated (where applicable).
A total of 11 models were considered in the reviewed literature. As for the IDEs they were integrated into, most of them use some combination of two of the most popular ones: Visual Studio Code and JetBrains (which includes IntelliJ IDEA and PyCharm).

3.2 RQ2: Techniques behind the models

In this subsection we will answer research question 2 by discussing the methods behind these code generation models established in RQ1. Although Table 2 contains all code generation tools mentioned in the selected literature, we will focus in this part on the ones that were the main subjects of the papers.
As a short overview, we notice a general preferences towards machine learning driven tools, with many of them using transformer models, such as the Generative Pre-trained Transformer 2 and 3 (GPT-2 and GPT-3). Other approaches include modeling information in the form of graphs or abstract syntax trees [8].
To begin with, IntelliCode [16] is a multilingual code completion tool based on a transformer model, defined in this paper as "a family of neural networks designed to process ordered sequential data". The transformer model used in this work is GPT-C, which is based on GPT-2, "an auto-regressive pre-trained model consisting of a decoder-only transformer

stack and one or more output layers". This is applied on "source code understanding", which they describe as a more constrained and rule-based form of Natural Language Understanding (NLU), in order to generate source code.

NL2Code [21] is a hybrid code generation and retrieval tool that uses Natural Language (NL) queries. The underlying code generation system uses a tree-based semantic parsing model, while the code retrieval is done using a wrapped search engine (in this case, Bing). The semantic parsing model used is the one by Xu et al. [20], which is an improved version of the one by Yin et al.[22], which uses a tree-based neural network to encode natural language and produce corresponding source code.

Another tool that is able to generate code from NL requests is HISyn [23]. This work aims to eliminate the need for large sets of labeled training examples in the learning phase, instead using a human inspired learning approach based on NLU. It is able to combine the information available about the APIs with its understanding of the NL user query to produce syntactically correct code, and as such requires no training data. The HISyn framework consists of three parts: a Domain Knowledge Constructor, a Front-End that turns the NL query into a dependency graph and a Back-End that uses grammar-graph-based translation on the previous results in order to generate source code.

In this analysis of machine learning based automatic code generation [24], three tools are described: AiXcoder, Kite and Intellij IDEA (the code completion tool in the IDE with the same name). Only two of them are integrated into IDEs, with Kite, a GPT-2 based code completion tool [8], functioning as a desktop application. AiXcoder is a deep-learning model capable of learning from a programmer's actions and defining patterns to improve its predictions. Intellij IDEA provides numerous ways of code completion, each of them with different underlying algorithms.

Another study of AI-driven code generation tools [1], analyses OpenAI Codex, DeepMind AlphaCode and Amazon CodeWhisperer. Codex is a descendant of the advanced Large Language Model (LLM) GPT-3 and it is capable of generating, explaining and translating code in multiple programming languages based on NL queries. It is also the model that powers GitHub Copilot. Both Codex and Alpha-Code use a transformer-based model and the two are quite similar over-all, with the latter being trained on more complex problems and, as such, in this study it is claimed that it is more equipped to handle them. Lastly, CodeWhisperer is also a similar tool, with the difference that it learns from the programmer's previous code and various comments in order to improve its predictions

3.3 RQ3: Indicators for evaluation

In this subsection we will answer research question 3 by presenting the indicators that have been used in the selected literature to evaluate the code generation models in question. Table 3 serves as a quick overview.

In this study of IntelliCode [16], one of the metrics used for evaluating the language model is *perplexity*, defined as
$PPL = exp(-\sum_i^t(P(x_i)logP(x_i))), \forall i \in 0...T,$

Indicator	Ref	Description
Perplexity	[16]	How much the model is "surprised" by new data
ROUGE	[16]	String similarity between suggestions and target code
Levenshtein similarity	[16]	How many edits does it take to transform suggestion into target code
Surfacing Rate (SR)	[16]	Total number of completions displayed / number of times a completion could be shown
Click-Through-Rate (CTR)	[16]	Accepted completions / total completions
BLEU accuracy score	[21; 4]	Token-level overlap between suggestion and reference solution
Accuracy	[10; 9; 17]	Fraction of times the correct code is suggested first
Precision	[13; 4]	Accuracy of positive predictions
Recall	[13; 4]	Completeness of positive predictions
F-measure	[13; 4]	Harmonic mean of recall and precision
Top-k accuracy	[5; 18; 6]	How often the correct solution appears in the first k recommendations
Mean reciprocal rank (MRR)	[5; 18; 6]	Overall rank of the result
Soundness	[14]	Syntactical correctness of suggestions
Completeness	[14]	Is the suggestion correct and complete enough to provide the desired code snippet
Performance	[14; 17]	How fast are the suggestions generated

Table 3: Indicators used to evaluate code generation models

where x_i is the truth label and $P(x_i)$ is the model output. Thus, it is aimed for lower perplexity, as the lower the perplexity is, the higher the probabilities assigned to the true tokens. The researchers then go on to measure offline performance by using the *Recall-Oriented Understudy for Gisting Evaluation (ROUGE)* and the *Levenshtein similarity*. For their online evaluation they collected anonymous usage data and measured the *surfacing rate (SR)* and the *click-through-rate (CTR)*. They define SR as "the total number of completions displayed divided by the total number of times a completion could potentially be shown, which is after every character typed into a code document when the extension is active" and the CTR as "the fraction of accepted completions over the total number of completions displayed".
The evaluation of NL2Code [21] is done using the *accuracy*, calculated with the *BLEU score*, which measures the similarity between the generated code and a reference implementation. On the other hand, in [10; 9], the accuracy is computed as the fraction of times when the correct code appears first in their list of suggestions. Also focusing on the order of the suggestions in a code completion model, this study [24], uses the average number of keys it takes to get to the desired piece of code in the suggestion box as an indicator for their model. Another metric used in the same paper is the length of the suggestion list, as it is argued that a larger variety of code completion suggestions offers programmers more flexibility and a higher chance of finding what they need.
In another study [13], the metrics of
$precision(P) = \frac{Recommendations_{made \cap relevant}}{Recommendations_{made}}$,
$recall(R) = \frac{Recommendations_{made \cap relevant}}{Recommendations_{relevant}}$
and $F - measure = \frac{2*P*R}{P+R}$
are claimed to be most used for evaluating a neural network model. We see the same metrics used in combination with the *BLEU score* in another study [4] In multiple papers [5; 18; 6] the *top-k accuracy* and *mean reciprocal rank (MRR)* are used as evaluation metrics. In this paper [14], the metrics used for evaluation are *soundness* - are the code suggestions semantically correct as to not introduce new errors, *completeness* - is there a suggested option that is correct and complete enough to provide at least the beginning of the desired code snippet and *performance* - how long does it take for the model to provide the suggestions.
For this neural code completion tool [17], the aspects of accuracy, model size and suggestion speed are used in the evaluation.

3.4 RQ4: Aspects to be considered

Here we will answer research question 4 by presenting the aspects that should be considered when designing an in-IDE code generation model, as suggested by the reviewed literature. We will then propose a set of summarized guidelines for this purpose.
In their work of improving code completion tools [2], when considering their constraints, Bibaev et al. consider the concepts of speed, reliability and available resources. Other aspects mentioned that are worth taking into account are whether internet access in required for the tool to function and data sensitivity and collection. Data privacy is especially important in the cases where the model learns from real users' behaviour.
In the context of discussing the benefits of deep learning models, this study [12] defines four essential characteristics of code generation. These are "automatic feature extraction and generation, capturing the sequential properties, end to end learning and generalizability of code segments". It is claimed that any model must make a trade-off of these characteristics.
In their exploration of prompt programming, Fiannaca et al. [3] had domain experts evaluate their prototype and give feedback. Of the most notable desired features are automatic support, error handling and automatic help in constructing better prompts.
In the user evaluation of their code generation plugin, these researchers [21] had programmers with different backgrounds solve a set of exercises using their model and then collected their feedback. They first found, by having an expert analyze their solutions and plugin usage, that most of the queries were not specific enough and that the generated code directly improves with the quality of the search query. In the user feedback, several participants express the need for some sort of documentation or explanation for the provided code suggestions, while many said they expected the plugin to be "smarter" and understand bits of information from the context. One last user suggestion was to make the plugin interactive so the model can request extra information when the user does not know to offer it.
Also mentioned in RQ3, but worth noting here are the aspects of soundness and completeness. One last study [17] also ponders on the use of resources, arguing about their tool's accessibility to people without internet or the latest technology.
As such, we propose the following set of summarized guidelines for designing in-IDE code generation tools:

- Code generation should be fast [2; 17].
- All suggestions should be sound and complete [14].
- The generated code should be explainable and provide documentation [3; 21].
- The suggested code segments should be generalizable [12].
- Code generation tools should provide automatic help and guidance for the user and be able to recover from errors [3; 21].
- The tools should be available with as little constraints as possible, such as internet access or high-end technology [17; 2].

4 Discussion

After reviewing the numerous models presented in the selected literature, we observed a growing development of AI-driven tools for turning concept into code, which appears to be the emerging trend in the most recent years. Several problems have been identified and, as such, improvements have been proposed in the studied papers. One of the greatest challenges in the use of these tools right now is teaching users how to compose natural language prompts in lack of a strict grammar [3]. In the same context, other struggles that were pointed out in the research were the need for a large set of labelled training data [23] and the need for code generation and code completion tools to learn and gather information from the context [10].

Although code completion has been a crucial IDE feature for quite some time now, new points of improvement in the field are still being researched. For example, Pelsmaeker et al. [14] claim that code completion models are often "either too generic or too specific" for multiple language support. Another thing that could be improved is the ordering of the suggestions, with the most relevant ones on top so the users do not have to scroll through an alphabetically order list [18; 6].

Considering this, we expect the next step in improving code generation and making it accessible to people with all different backgrounds and knowledge levels, to be in enhancing the way the user interacts with the model. This entails work both on the programming part, but also on the human users' part. On one hand, code generation tools should aim to be interactive, tolerant to errors and display conversational memory, while also providing explanations and reasoning for their suggestions. Even with all these characteristics, if we expect such tools to be capable of everyday use by people who are not in the computer science field, it is crucial we teach them how to interact with them. Things like how to construct a proper query, what key words to use and how much context to provide make a significant difference in the quality of the suggestions, but are not available knowledge to everyone.

When looking at the underlying code generation techniques, we see that machine learning based code generation is the most prominent approach. More specifically, many of the models we considered in this review rely of some variant of the GPT model, which has become even more popular recently, with the apparition of ChatGPT. One disadvantage of models like these is that they require significant amounts of training data, which motivates the emerging research [23] on other ways of learning that avoid this.

4.1 Limitations

As discussed in the methodology, due to a large number of available papers over a very large period of time, we decided to limit our search to only the last 5 years, so works dated before 2019 were excluded from this literature review. Another limitation is the fact that, due to time constraints, only a limited number of papers (19) could be reviewed in the course of this research. To that extent, the code generation models, evaluation techniques and general guidelines considered in this research are only the ones presented in the selected literature and are not a complete representation of all the available options. Furthermore, due to the nature of this project, the whole search and analysis were conducted by a single person. This is a limitation in and of itself when compared to a collaboration that can assure higher levels of objectiveness, coverage of the literature and quality control.

4.2 Responsible Research

Since this research does not deal with any humans or experiments, the types of bias that can arise are limited. In the context of a literature review, bias can arise mainly in the selection of the included papers, in the form of selection bias, in the case when the researcher purposely selects literature that supports their hypothesis, and excludes papers that contradict it. To make sure the selection process is as unbiased as possible, it follows strict steps with predefined inclusion and exclusion criteria and all results that fulfilled all the required characteristics were included in the study. Another possible type of bias that the researcher does not have control over is publication bias, that refers to the tendency to only publish those papers that present positive results. This literature review only considers works that have been published and peer-reviewed, so it could be subject to this bias, but at the same time it can be argued that only the code generation models that have been successful are relevant for analyzing their desired characteristics and synthesizing a set of guidelines, which is the main purpose of this research.

In order to assure the reproducibility and transparency of the used method, all steps were systematically recorded in the the methodology section. This includes the search query that was used, the databases that were sampled, the dates when the searches were performed and all the inclusion and exclusion criteria used in the filtering process. As such, with the provided information, anyone can reproduce the steps that yielded the 19 papers included in the study.

4.3 Threats to validity

We used the following study [25] as a guide to the different types of threats to validity encountered in a systematic literature review and reflected on how they were handled in this research. We will mention here only the applicable ones. In the planning phase, two possible threats to validity are having *incomprehensive venues or databases* and *restricted time span*. As it was mentioned before, due to time constraints, only a limited number of papers could be feasibly reviewed in the given time, so 3 representative databases were selected, excluding as such possible new results that would have been discovered if more databases could have been covered. A possible *culture bias* could be considered the fact that only papers in English were considered, due to the linguistical abilities of the researchers. In the conducting phase, a possible threat to validity that has been explained in the previous section is *publication bias*.

5 Conclusion

In this work we have analyzed the subject of in-IDE code generation models, identifying a set of example tools and presenting the techniques behind them and the indicators used in evaluating them. Furthermore, we have presented the aspects that should be considered when designing in-IDE code generation models and proposed a set of summarized guidelines from the reviewed literature for this purpose. Lastly, we have highlighted some of the issues with the current methods and the proposed improvements in emerging research.

As far as future research goes, we would like to see a larger-scale version of this study, without the time constraints, so that a greater selection of papers can be included and a more comprehensive list of code generation models can be studied. Furthermore, another opportunity for future work would be to develop a new code generation tool following the guidelines presented in this research.

References

[1] Brett A. Becker, Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, James Prather, and Eddie Antonio Santos. Programming is hard-or at least it used to be: Educational opportunities and challenges of ai code generation. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, pages 500–506, 2023.

[2] Vitaliy Bibaev, Alexey Kalina, Vadim Lomshakov, Yaroslav Golubev, Alexander Bezzubov, Nikita Povarov, and Timofey Bryksin. All you need is logs: improving code completion by learning from anonymous ide usage logs. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2022.

[3] Alexander J. Fiannaca, Chinmay Kulkarni, Carrie J. Cai, and Michael Terry. Programming without a programming language: Challenges and opportunities for designing developer tools for prompt programming. In *Extended Abstracts of the 2023 CHI Conference on Human Factors in Computing Systems*, pages 1–7, 2023.

[4] H. Hu, Q. Chen, and Z. Liu. Code generation from supervised code embeddings. *Communications in Computer and Information Science*, 1142 CCIS:388–396, 2019.

[5] Yasir Hussain, Zhiqiu Huang, Yu Zhou, and Senzhang Wang. Deepvs: an efficient and generic approach for source code modelling usage. *ELECTRONICS LETTERS*, 56(12):604–606, 2020.

[6] Maliheh Izadi, Roberta Gismondi, and Georgios Gousios. Codefill. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, 2022.

[7] Khalid S Khan, Regina Kunz, Jos Kleijnen, and Gerd Antes. Five steps to conducting a systematic review. *Journal of the royal society of medicine*, 96(3):118–121, 2003.

[8] K. T. Le, G. Rashidi, and A. Andrzejak. A methodology for refined evaluation of neural code completion approaches. *Data Mining and Knowledge Discovery*, 37(1):167–204, 2023.

[9] Fang Liu, Ge Li, Bolin Wei, Xin Xia, Zhiyi Fu, and Zhi Jin. A unified multi-task learning model for ast-level and token-level code completion. *Empirical Software Engineering*, 27(4), 2022.

[10] Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. Multi-task learning based pre-trained language model for code completion. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2020.

[11] Gail Murphy, Mik Kersten, and Leah Findlater. How are java software developers using the eclipse ide? *IEEE Software*, 23:76–83, 07 2006.

[12] A.R. M. Nizzad and Samantha Thelijjagoda. Designing of a voice-based programming ide for source code generation: A machine learning approach. In *2022 International Research Conference on Smart Computing and Systems Engineering (SCSE)*, volume 5, pages 14–21, 2022.

[13] Hayatou Oumarou and Ousmanou Dahirou. A novel code completion strategy. *International Journal of Advanced Computer Science and Applications*, 13(5):866–871, 2022.

[14] Daniel A. A. Pelsmaeker, Hendrik van Antwerpen, Casper Bach Poulsen, and Eelco Visser. Language-parametric static semantic code completion. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1):1–30, 2022.

[15] Hannah Snyder. Literature review as a research methodology: An overview and guidelines. *Journal of Business Research*, 104:333–339, nov 2019.

[16] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 1433–1443, New York, NY, USA, 2020. Association for Computing Machinery.

[17] Alexey Svyatkovskiy, Sebastian Lee, Anna Hadjitofi, Maik Riechert, Juliana Vicente Franco, and Miltiadis Allamanis. Fast and memory-efficient neural code completion. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021.

[18] Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. Pythia: Ai-assisted code completion system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery &amp Data Mining*. ACM, 2019.

[19] Z. Wang, F. Liu, Y. Hao, and Z. Jin. Adacomplete: improve dl-based code completion method's domain adaptability. *Automated Software Engineering*, 30(1), 2023.

[20] F. F. Xu, Z. Jiang, P. Yin, B. Vasilescu, and G. Neubig. Incorporating external knowledge through pre-training for natural language to code generation. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, pages 6045–6052, 2020.

[21] F. F. Xu, B. Vasilescu, and G. Neubig. In-ide code generation from natural language: Promise and challenges. *ACM Transactions on Software Engineering and Methodology*, 31(2), 2022.

[22] P. Yin and G. Neubig. A syntactic neural model for general-purpose code generation. In *ACL 2017 - 55th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference (Long Papers)*, volume 1, pages 440–450, 2017.

[23] Mitchell Young, Zifan Nan, and Xipeng Shen. Ide augmented with human-learning inspired natural language programming. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 110–114, 2022.

[24] Xiaojiang Zhang, Ying Jiang, and Zhijun Wang. Analysis of automatic code generation tools based on machine learning. In *2019 IEEE International Conference on Computer Science and Educational Informatization (CSEI)*, pages 263–270, 2019.

[25] Xin Zhou, Yuqin Jin, He Zhang, Shanshan Li, and Xin Huang. A map of threats to validity of systematic literature reviews in software engineering. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, pages 153–160, 2016.