Teaching Code LLMs to Use Autocompletion Tools in Repository-Level Code Generation

CHONG WANG, Nanyang Technological University, Singapore
JIAN ZHANG*, Nanyang Technological University, Singapore
YEBO FENG, Nanyang Technological University, Singapore
TIANLIN LI, Nanyang Technological University, Singapore
WEISONG SUN, Nanyang Technological University, Singapore
YANG LIU, Nanyang Technological University, Singapore
XIN PENG, Fudan University, China

Recent code large language models (LLMs) have shown promising performance in generating standalone functions. However, they face limitations in repository-level code generation due to their lack of awareness of *repository-level dependencies* (*e.g.,* user-defined attributes), resulting in *dependency errors* such as undefined-variable and no-member errors. In this work, we introduce ToolGen, an approach that integrates autocompletion tools into the code LLM generation process to address these dependencies. ToolGen comprises two main phases: Trigger Insertion and Model Fine-tuning (Offline), and Tool-integrated Code Generation (Online). During the offline phase, ToolGen augments functions within a given code corpus with a special mark token, indicating positions to trigger autocompletion tools. These augmented functions, along with their corresponding descriptions, are then used to fine-tune a selected code LLM. In the online phase, ToolGen iteratively generates functions by predicting tokens step-by-step using the fine-tuned LLM. Whenever a mark token is encountered, ToolGen invokes the autocompletion tool to suggest code completions and selects the most appropriate one through constrained greedy search.

We conduct comprehensive experiments to evaluate ToolGen's effectiveness in repository-level code generation across three distinct code LLMs: CodeGPT, CodeT5, and CodeLlama. To facilitate this evaluation, we create a benchmark comprising 671 real-world code repositories and introduce two new dependency-based metrics: *Dependency Coverage* and *Static Validity Rate*. The results demonstrate that ToolGen significantly improves *Dependency Coverage* by 31.4% to 39.1% and *Static Validity Rate* by 44.9% to 57.7% across the three LLMs, while maintaining competitive or improved performance in widely recognized similarity metrics such as BLEU-4, CodeBLEU, Edit Similarity, and Exact Match. On the CoderEval dataset, ToolGen achieves improvements of 40.0% and 25.0% in test pass rate (Pass@1) for CodeT5 and CodeLlama, respectively, while maintaining the same pass rate for CodeGPT. ToolGen also demonstrates high efficiency in repository-level code generation, with latency ranging from 0.63 to 2.34 seconds for generating each function. Furthermore, our generalizability evaluation confirms ToolGen's consistent performance when applied to diverse code LLMs, encompassing various model architectures and scales.

CCS Concepts: • **Software and its engineering → Automatic programming**.

Additional Key Words and Phrases: repository-level code generation, code LLMs, tool integration

*Corresponding Author: Jian Zhang

Authors' addresses: Chong Wang, Nanyang Technological University, Singapore; Jian Zhang, Nanyang Technological University, Singapore; Yebo Feng, Nanyang Technological University, Singapore; Tianlin Li, Nanyang Technological University, Singapore; Weisong Sun, Nanyang Technological University, Singapore; Yang Liu, Nanyang Technological University, Singapore; Xin Peng, Fudan University, China.

ACM 1557-7392/2025/1-ART
https://doi.org/10.1145/3714462

1 INTRODUCTION

Code generation has been a longstanding focal point in the field of software engineering. Recent advancements have introduced a variety of code large language models (LLMs) [7, 12, 13, 15, 17, 21, 28, 29, 33, 37, 46, 53, 54] constructed upon the Transformer model architecture [45], achieving promising performance in code-related applications [19, 20, 47–50, 52, 57, 59, 60]. These models are either pre-trained or fine-tuned on extensive code corpora, enabling them to automatically generate code based on provided natural language descriptions. These code LLMs have demonstrated notable effectiveness in the generation of code blocks or functions. For instance, CodeLlama [37], built upon the foundational Llama2 model [43], has achieved state-of-the-art results among open code LLMs (*e.g.,* CodeGen [33] and StarCoder [28]), on benchmarks like HumanEval [15] and MBPP [9] that focus on standalone functions.

However, it is crucial to emphasize that in real-world code repositories, more than 70% of functions are not standalone [56]. Code LLMs encounter significant challenges when generating such real-world functions, primarily because they cannot be aware of ***repository-level dependencies***, such as user-defined functions and attributes, during the code generation process [56]. This limitation often leads to the generation of code with ***dependency errors***, including *undefined-variable* and *no-member* errors. These errors impede the usability and effectiveness of the code LLMs [44]. For example, consider the scenario depicted in Figure 1. A code LLM (*e.g.,* CodeLlama) might incorrectly predict "`_updates`" after generating "`... self.`", resulting in a *no-member* error because the object "`self`" does not possess an attribute named "`_updates`".

Meanwhile, modern Integrated Development Environments (IDEs) take a different approach, which typically incorporates code autocompletion tools based on program analysis. These tools, like Jedi [2], leverage their ability to analyze the current incomplete function's state and project context to provide ***valid*** completion recommendations. This includes suggestions for accessible variables, attributes, and functions. For instance, when encountering "`self.`" in Figure 1, Jedi can infer and recommend 68 accessible attributes defined within "`self`", including the target suggestion "`_registered_updates`". Therefore, if we can seamlessly switch between code LLMs and the use of autocompletion tools, we have the potential to significantly reduce the occurrence of dependency errors in repository-level code generation.

In fact, recent research has delved into the integration of external tools into the generation process of LLMs to mitigate their limitations in constrained generation scenarios. One noteworthy example is ToolFormer [38], which creates an augmented dataset to instruct LLMs on invoking existing arithmetic calculators. This integration effectively reduces errors in generated text involving arithmetic calculations. Building upon ToolFormer's inspiration, Zhang *et al.* [61] introduce ToolCoder, an approach designed to teach LLMs how to utilize *information-retrieval-based* (IR-based) API search tools during the code generation process. While ToolCoder targets the generation of functionally correct standalone functions and demonstrates promising results, the integrated IR-based API search tools do not consider repository-level dependencies, limiting their potential in resolving dependency errors. Additionally, ToolFormer and ToolCoder are unable to handle scenarios where the tools return multiple candidates. Another relevant example of harnessing external tools is Repilot [55], which leverages code completion tools to filter out impractical suggestions made by LLMs in the context of automatic program repairing (APR). Unlike repository-level code generation, Repilot's primary focus is on generating valid *single-hunk bug-fix patches* rather than entire functions. When applying Repilot to function-level code generation, the autocompletion tools are frequently triggered unnecessarily, resulting in significant overhead and impracticality. Despite these limitations, these works provide a solid starting point for the integration of external tools.

In this work, we aim at integrating program-analysis-based code autocompletion tools into the generation process of code LLMs. Achieving the incorporation presents two key challenges. ***(i) Determining when to trigger the invocation of autocompletion tools during the generation process:*** The generation process of LLMs is a step-by-step decoding process where each subsequent token is predicted based on previous tokens.

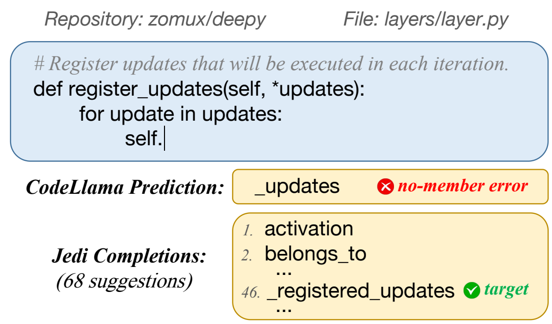


Fig. 1. Illustrative Example of LLM Prediction and Tool Completion

In general, a function consists of dozens or even hundreds of tokens, making it impractical to invoke code autocompletion tools at every decoding step. In the case of tools like ToolFormer and ToolCoder, ChatGPT is employed to augment the training corpus by introducing special tokens into the text or code to mark positions where tool invocation is needed. After training on this augmented corpus, LLMs can predict the special token at the appropriate step, thereby triggering tool invocation. However, this ChatGPT-based augmentation method is less effective for repository-level code generation due to the presence of repository-level dependencies. The special token must be precisely inserted at positions involving such dependencies, such as when accessing user-defined variables. ***(ii) Selecting the target suggestion from the recommended completions of autocompletion tools:*** Different from tools like arithmetic calculation or API search integrated into ToolFormer and ToolCoder, which return a single result for each invocation, autocompletion tools often provide multiple completion suggestions sorted alphabetically. For instance, as depicted in Figure 1, Jedi returns a list of 68 completion suggestions (excluding builtin attributes), with the target suggestion being the 46th one in the list. Consequently, after invoking autocompletion tools, it is essential to assess the suggestions based on the generated code and select the most appropriate one. Furthermore, this selection process needs to be seamlessly integrated into the code generation process to ensure efficiency and coherence.

To tackle the challenges, we propose ToolGen, an approach to integrate autocompletion tools into the generation process of code LLMs to support repository-level code generation. ToolGen has two main phases: Trigger Insertion and Model Fine-tuning (Offline), and Tool-integrated Code Generation (Online). In the offline phase, ToolGen analyzes source files within a corpus of code repositories, creating abstract syntax trees (ASTs) and extracting function definitions. It augments these functions by inserting a special token, `<COMP>`, signifying the positions to trigger autocompletion tools. The insertion positions are established by navigating through the functions and identifying the identifiers that can be recommended by autocomplete tools. These augmented functions, paired with their respective descriptions, are then employed to fine-tune a selected code LLM. In the online phase, ToolGen iteratively constructs a function based on a provided description by predicting tokens step-by-step through the fine-tuned LLM. Whenever a `<COMP>` token is encountered, ToolGen invokes the autocompletion tool to suggest code completions, drawing from the current repository context. Subsequently, it identifies the most appropriate suggestion through a constraint greedy search algorithm, appending this selected suggestion to the current tokens. This process continues as it predicts tokens until a specified termination condition is satisfied.

We conduct extensive experiments to evaluate the effectiveness of ToolGen in repository-level code generation across three distinct code LLMs, namely, CodeGPT [31], CodeT5 [54], and CodeLlama [37]. To facilitate this evaluation, we first construct a benchmark, which includes 12,406 Python functions from 671 real-world code repositories and 176 coding tasks from CoderEval dataset [56]. We define two new repository-level metrics, namely

Dependency Coverage and *Static Validity Rate*. *Dependency Coverage* quantifies the proportion of repository-level dependencies present in ground-truth functions and successfully covered by the generated functions, while *Static Validity Rate* measures the percentage of generated functions that pass a dependency error check. The evaluation results on the 12,406 functions demonstrate that ToolGen exhibits comparable or improved performance in widely-recognized similarity metrics such as BLEU-4, CodeBLEU, Edit Similarity, and Exact Match. Importantly, ToolGen achieves significant improvements in *Dependency Coverage*, ranging from 31.4% to 39.1%, and *Static Validity Rate*, spanning from 44.9% to 57.7%, across the three code LLMs. On the 176 tasks derived from CoderEval, ToolGen achieves improvements of 40.0% and 25.0% in test pass rate (Pass@1) for CodeT5 and CodeLlama, respectively, while maintaining the same pass rate for CodeGPT. ToolGen also demonstrates high efficiency in repository-level code generation, with average latency ranging from 0.63 to 2.34 seconds, attributed to offline fine-tuning with trigger insertion. Moreover, the results from our generalizability evaluation confirm that ToolGen consistently performs well across a variety of code LLMs, with different model architectures and scales.

In summary, this paper presents the following key contributions:

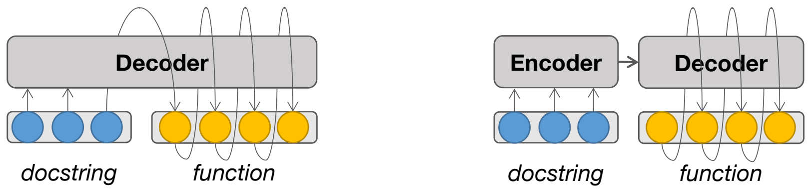
- **ToolGen**, an approach that seamlessly integrates autocompletion tools into the generation process of code LLMs, which consists of Trigger Insertion and Model Fine-tuning (Offline), and Tool-integrated Code Generation (Online). ToolGen seamlessly integrates the autocompletion tool into the generation process of code LLMs, thereby enhancing repository-level code generation. The offline phase results in an **Augmented Dataset**, which comprises 249,298 Python functions sourced from a diverse selection of 12,231 code repositories. Each function is augmented with a special token, <COMP>, which signifies positions suitable for invoking autocompletion tools.
- **An Evaluation Benchmark**, which encompasses 12,406 Python functions drawn from 671 real-world code repositories and 176 coding tasks with test cases derived from CoderEval, along with the introduction of two novel repository-level metrics: *Dependency Coverage* and *Static Validity Rate*.
- **Extensive Experimental Results**, which affirm the efficacy of ToolGen in repository-level code generation. ToolGen demonstrates substantial improvements in *Dependency Coverage*, ranging from 31.4% to 39.1%, and *Static Validity Rate*, spanning from 44.9% to 57.7%, across three distinct code LLMs. Additionally, ToolGen achieves 40% and 25% improvements in test pass rate for CodeT5 and CodeLlama, respectively, with high generation efficiency.

2 PRELIMINARIES

2.1 Code LLMs

Typically, there are two main categories of code LLMs that can be employed for code generation. These categories include decoder-only models and encoder-decoder models, each of which conducts the code generation process base on a given description as outlined below:

- **Decoder-only Models:** Illustrated in Figure 2a, decoder-only code LLMs, such as CodeGPT [31] and CodeLlama [37], consist solely of a decoder component derived from the Transformer architecture [45]. An employed decoder-only model first tokenizes the input description into a sequence of tokens. Subsequently, it feeds this token sequence into the model's decoder and proceeds to predict a function token-by-token, based on the context provided by the description and previously predicted tokens.
- **Encoder-Decoder Models:** As depicted in Figure 2b, encoder-decoder code LLMs, such as CodeT5 [54] and CodeT5+ [53], encompass both the encoder and decoder components of the Transformer architecture. In this case, the employed model also tokenizes the description into a token sequence, but the sequence is first processed by the model's encoder. The model's decoder is then tasked with predicting a function token-by-token, relying on the representation produced by the encoder and the context provided by the preceding tokens.



(a) Decoder-only Model (b) Encoder-Decoder Model

Fig. 2. Decoder-only Model and Encoder-Decoder Model

On top of the standard generation process, to ensure that the employed code LLM can recognize and predict the special token `<COMP>`, we initially incorporate this token into the LLM's vocabulary, denoted as $\mathbb{V}_{llm}$. Formally, this addition results in an expanded vocabulary represented as:

$$\mathbb{V} \leftarrow \mathbb{V}_{llm} \cup \{\texttt{<COMP>}\} \tag{1}$$

For the employed code LLM, within the generation process, we define its tokenization process as a procedure:

$$\textsc{llm-tokenize} : \Sigma^*_{char} \longrightarrow \mathbb{V}^* \tag{2}$$

Here, Σ^*_{char} represents a character sequence of either a description or a code snippet, and $\mathbb{V}^*$ corresponds to the resulting sequence of tokens drawn from $\mathbb{V}$.

The next token prediction involved in each step is defined as a procedure:

$$\textsc{llm-predict} : (\mathbb{V}^*, \mathbb{V}^*) \longrightarrow [0,1]^{|\mathbb{V}|} \tag{3}$$

In this context, the two input token sequences ($\mathbb{V}^*$) represent a description and an incomplete function, respectively, while $[0,1]^{|\mathbb{V}|}$ signifies a probability distribution encompassing $|\mathbb{V}|$ probabilities $[0,1]$. Here, $|\mathbb{V}|$ is the size (token numbers) of the vocabulary $\mathbb{V}$.

Example: In Figure 1, CodeLlama takes the description "Register updates..." and the incomplete function "... `self.`" as inputs. It then performs a prediction, generating a probability distribution of size $|\mathbb{V}|$, wherein the token "`_updates`" exhibits the highest probability among all tokens.

2.2 Autocompletion Tools

An autocompletion tool takes a code repository and a caret position (defined as a tuple containing source file, line number, and column number) as input and provides a list of completion suggestions. We define this completion process as a procedure:

$$\textsc{tool-complete} : (\Sigma_{repo}, \Sigma_{pos}) \longrightarrow \Sigma^*_{iden} \tag{4}$$

Here, Σ_{repo} and Σ_{pos} respectively represent the domains of code repositories and caret positions, Σ_{iden} encompasses all possible identifiers such that Σ^*_{iden} is a list of identifiers. It's worth noting that autocompletion tools often provide a wide range of completion suggestions, including keywords and partial identifiers. In our context, we focus solely on identifier-level completions, as keywords are relatively straightforward for code LLMs to predict, and partial identifiers are encompassed by identifier-level completions.

Example: In Figure 1, when provided with the code repository and caret position, Jedi is capable of generating 86 completion suggestions for the incomplete function "... `self.`".

3 APPROACH

In this section, we elaborate on our approach named ToolGen to integrate autocompletion tools into the generation process of code LLMs to support repository-level code generation.

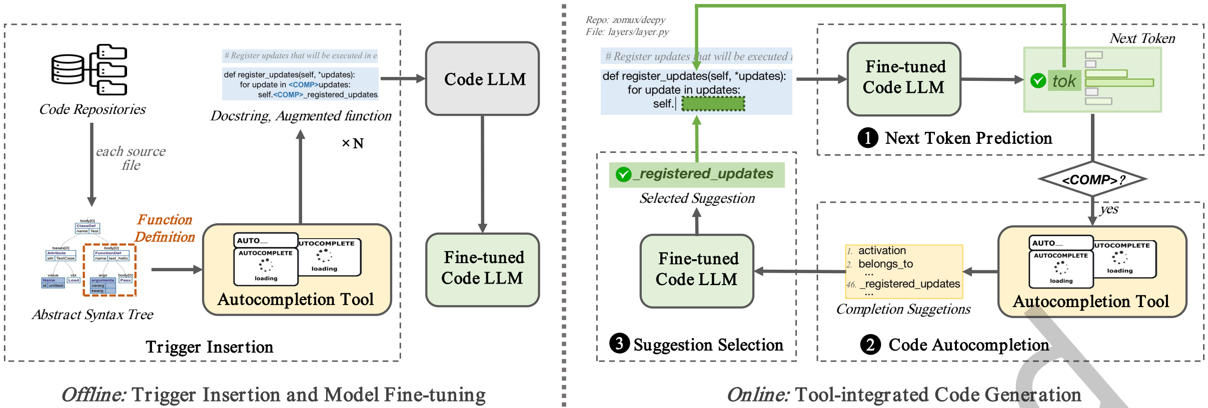


Fig. 3. Approach Overview of ToolGen

3.1 Overview

Figure 3 presents an overview of ToolGen, which consists of two main phases, namely (i) **Trigger Insertion and Model Fine-tuning** (Offline) and (ii) **Tool-integrated Code Generation** (Online).

In trigger insertion and model fine-tuning, ToolGen parses each source file in the given code repositories into an abstract syntax tree (AST) and then extracts function definitions from the AST; For each extract function definition, ToolGen then utilizes an autocompletion tool to augment it with the special token `<COMP>` to mark the positions to invoke the tool, and then assembles a pair of description and augmented function; After process all code repositories, ToolGen employs the resulting pairs of descriptions and augmented functions to fine-tune a code LLM, resulting in a fine-tuned code LLM that can predict `<COMP>` at suitable positions to trigger the autocompletion tool.

In tool-integrated code generation, ToolGen generates a token sequence to form a function by an iterative process, in which, at each step, one or multiple tokens are yielded by the fine-tuned code LLM and the employed autocompletion tool. At certain step, ❶ the fine-tuned code LLM takes the given description and the incomplete function as inputs and predicts the next token; The predicted token is appended to the incomplete function; ❷ If the predicted token equals `<COMP>`, the autocomplete tool is triggered and a list of completion suggestions is returned based on the current repository context; ❸ ToolGen then selects the most suitable one from the suggestions with the fine-tuned code LLM and appends the selected suggestion to the incomplete function.

3.2 Trigger Insertion and Model Fine-tuning

3.2.1 Trigger Insertion. We employ a trigger insertion method to facilitate the learning process of code LLMs in determining when to utilize autocompletion tools during code generation. In this method, the special token `<COMP>` is inserted at specific locations within code functions, indicating when autocompletion tools should be triggered.

Given a code repository $\mathcal{R}$, we traverse each source file $file$ within it based on the file's suffix (*e.g.,* `.py` for Python) and then proceed to analyze the functions defined in the source file. To achieve this, we parse the source file into an abstract syntax tree (AST), where the functions are represented as function-definition nodes. Each function-definition node contains multiple AST-tokens, which are smallest individual units, such as keywords, identifiers, literals, operators, and punctuators, within programming language syntax. Note that these AST-tokens differ from the tokens in the LLM's vocabulary $\mathbb{V}_{llm}$. Typically, an AST-token comprises one or more tokens from

$\mathbb{V}_{llm}$. For example, the AST-token "`_registered_updates`" consists of six tokens in vocabulary of CodeLlama, *i.e.,* ["`_`", "`register`", "`ed`", "`_`", "`up`", "`dates`"].

For each function within the source file, we identify its corresponding function-definition node, denoted as *node*, and apply Algorithm 1 to it. The purpose of this algorithm is to traverse the function body and identify specific identifiers that are eligible for suggestions by autocomplete tools. Subsequently, the special token `<COMP>` is inserted in front of these chosen identifiers. More specifically, as the algorithm iterates through each AST-token t within the function body $node.body$ (line 2), it performs two crucial checks. First, it employs the isIdentifier procedure to determine whether t is an identifier. Second, it verifies that t is not a built-in attribute, such as "`__dict__`" in Python, using the isBuiltin procedure. These conditions are essential because dependency errors often arise from user-defined attributes categorized as identifiers rather than other AST-tokens like language keywords. Additionally, these checks prevent the insertion of `<COMP>` at positions where the code LLM can confidently predict the following tokens, thus minimizing unnecessary tool invocations. When both conditions are met, the algorithm updates the caret position $\mathcal{P}$ to the start position of t (line 4) and invokes the autocompletion tool to obtain a list of completion suggestions, denoted as $\mathbb{C}$ (line 5). If $\mathbb{C}$ contains t, indicating that the tool can propose the desired identifier, the special token `<COMP>` is inserted before t to mark the position for triggering the autocompletion tool (lines 6-7). Upon executing the algorithm, we obtain the augmented function code $\mathcal{F}_{aug}$.

Algorithm 1: Trigger Insertion

Input: Repository $\mathcal{R}$, Source File $file$, Function-definition node $node$
Output: Augmented function $\mathcal{F}_{aug}$

1 $\mathcal{F}_{aug} \leftarrow$ getSignature($node$) // signature
2 **for** t ***in*** $node.body$ **do**
3   **if** *isIdentifier*(t) **and not** *isBuiltin*(t) **then**
4         $\mathcal{P} = (file, t.start\_line, t.start\_column)$
5         $\mathbb{C} \leftarrow$ tool-complete($\mathcal{R}, \mathcal{P}$)
6   **if** $t \in \mathbb{C}$ **then**
7             $\mathcal{F}_{aug}.append($`<COMP>`$)$
8         $\mathcal{F}_{aug}.append(t)$

Next, we assemble a tuple $(\mathcal{D}, \mathcal{F}_{aug})$, in which $\mathcal{D}$ corresponds to the concatenation of the signature and docstring of the parsed function. Note that functions lacking corresponding docstrings are omitted from our process as our repository-level code generation relies on textual descriptions as input. Once we complete the processing of all code repositories, we accumulate an ***augmented dataset*** that contains a substantial number of these data tuples.

Note that our trigger insertion method can be applied to arbitrary code and is not limited to function bodies alone. Currently, we focus exclusively on function bodies, as our primary application scenario involves generating code based on the given natural language descriptions. Extracting descriptions for code blocks outside functions for model training and evaluation is challenging, due to the difficulty in determining the scope of line comments [14, 24]. Therefore, we solely consider function bodies, where corresponding descriptions can be readily obtained from function docstrings.

Example: In Figure 4, we showcase an augmented function that contains four instances of the special token `<COMP>`. These tokens have been inserted at positions where the desired identifiers, namely "`updates`", "`_registered_updates`", "`add`", and "`update`", are found within the suggestion lists of the autocompletion tool.

```
def register_updates(self, *updates):
    for update in <COMP>updates:
        self.<COMP>_registered_updates.<COMP>add(<COMP>update)
```

Fig. 4. Augmented Function

3.2.2 Model Fine-tuning. During the fine-tuning process, we supply the collected descriptions and augmented functions to optimize the parameters of the employ code LLM (base model), adhering to established practices in code generation tasks. Specifically, for each pair consisting of a description $\mathcal{D}$ and an augmented function $\mathcal{F}_{aug}$, both are tokenized into sequences of tokens and subsequently fed into the base model to undergo the token-by-token generation process described in Section 2.1. At each step, a cross-entropy loss is computed between the predicted probability distribution of the next token and the ground-truth next token present in $\mathcal{F}_{aug}$.

In the case of code LLMs with an extensive number of parameters, such as CodeLlama-7B with 7 billion parameters, fine-tuning all parameters becomes computationally challenging due to resource limitations. To address this, we employ Low-Rank Adaptation (LoRA) [23] as a parameter-efficient fine-tuning technique. LoRA relies on low-dimensional representations and a freeze-and-inject strategy, where the majority of the model parameters remain fixed, and trainable low-rank matrices are introduced into specific transformer layers, particularly the projection matrices within the attention module, to approximate weight updates.

3.3 Tool-integrated Code Generation

Based on the fine-tuned code LLM and the employed autocompletion tool, we perform a tool-integrated code generation process that is aware to the repository-level dependencies.

3.3.1 Overall Process. Algorithm 2 outlines the overall tool-integrated generation process, comprising three crucial parts based on the fine-tuned code LLM and the employed autocompletion tool: ❶ Next Token Prediction, ❷ Code Autocompletion, and ❸ Suggestion Selection. This algorithm takes a code repository $\mathcal{R}$, an insertion position $\mathcal{P}$, and a description $\mathcal{D}$ as inputs and follows an iterative process to generate a token sequence, ultimately constructing a function denoted as $\mathcal{F}$. Here, the tokens are drawn from the expanded vocabulary $\mathbb{V}$ defined in Equation 1.

The iterative process commences with the `<BOS>` token (representing the beginning of the sequence), *i.e.,* $\mathcal{F} \leftarrow$ [`<BOS>`] in line 2, and proceeds by iteratively updating $\mathcal{F}$ until it reaches the `<EOS>` token (representing the end of the sequence). During each iteration step, the algorithm utilizes the description $\mathcal{D}$ and the current incomplete function $\mathcal{F}$ as inputs for the fine-tuned code LLM to execute the LLM-PREDICT procedure. This procedure predicts a $|\mathbb{V}|$-dimension probability distribution $\boldsymbol{p}^{|\mathbb{V}|}$ for the tokens in the vocabulary $\mathbb{V}$ (line 4). Subsequently, the token tok with the highest probability is selected using the commonly used ARGMAX function [1] (line 5). The selected token tok is then appended to $\mathcal{F}$ (line 6). If tok corresponds to the `<EOS>` token, the iterative process concludes, yielding the final generated function (lines 7-8).

If tok corresponds to the special token `<COMP>`, the autocompletion tool is triggered to provide a list of completion suggestions denoted as $\mathbb{C}$. These suggestions are produced based on the code repository $\mathcal{R}$ and the caret position $\mathcal{P}'$ after inserting $\mathcal{F}$ at $\mathcal{P}$ (lines 9-11). Notably, when $\mathcal{F}$ is inserted using the INSERT procedure, any `<COMP>` tokens within it are removed to prevent syntax errors. The fine-tuned code LLM is then employed to assess the completion suggestions and select the most suitable one for $\mathcal{F}$ by the **LLM-SELECT** procedure (line 12). The tokens from the selected suggestion are concatenated to $\mathcal{F}$.

Example: In the case of the incomplete code snippet shown in Figure 1, Algorithm 2 predicts the next token as `<COMP>` through the fine-tuned code LLM. This prediction triggers the autocompletion tool. Subsequently, the

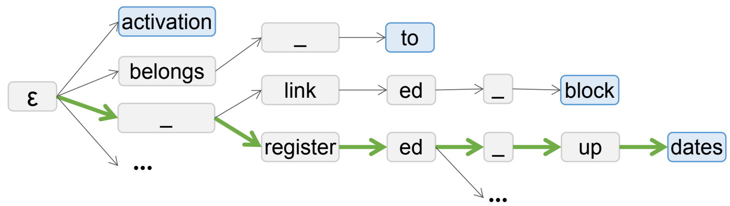


Fig. 5. Example Prefix Tree

resulting completion suggestions are fed into the LLM-SELECT procedure, which determines the most appropriate suggestion.

Algorithm 2: Tool-integrated Code Generation

Input: Repository $\mathcal{R}$, Description $\mathcal{D}$, Insertion Position $\mathcal{P}$
Output: Function $\mathcal{F}$

$\mathcal{D} \leftarrow$ LLM-TOKENIZE($\mathcal{D}$)
$\mathcal{F} \leftarrow$ [<BOS>]
while *true* **do**
    /* ❶ Next Token Prediction */
    $\boldsymbol{p}^{|\mathbb{V}|} \leftarrow$ LLM-PREDICT($\mathcal{D}, \mathcal{F}$)
    $tok \leftarrow$ ARGMAX($\mathbb{V}, \boldsymbol{p}^{|\mathbb{V}|}$)
    $\mathcal{F} \leftarrow \mathcal{F} \oplus [tok]$
    if tok = <EOS> **then**
        break
    /* ❷ Code Autocompletion */
    if tok = <COMP> **then**
        $\mathcal{P}' \leftarrow$ INSERT($\mathcal{P}, \mathcal{F}$)
        $\mathbb{C} \leftarrow$ TOOL-COMPLETE($\mathcal{R}, \mathcal{P}'$)
        /* ❸ Suggestion Selection */
        LLM-SELECT($\mathcal{D}, \mathcal{F}, \mathbb{C}$)

3.3.2 Completion Suggestion Selection. Algorithm 3 provides a description of the LLM-SELECT procedure, which is called within Algorithm 2. To begin, it tokenizes each completion in $\mathbb{C}$ into a sequence of tokens from $\mathbb{V}$ using the code LLM's tokenizer (via the LLM-TOKENIZE procedure) and inserts this token sequence into a prefix tree [5], denoted as *trie* (lines 1-5). Each node in the tree possesses four properties: *node.token*, *node.tok_idx*, *node.children*, and *node.is_terminal*, indicating the token stored in the node, the index of the stored token in $\mathbb{V}$, the child nodes of the current node, and whether the node corresponding to the terminal of a token sequence. The root node, *trie.root*, is a unique node that stores ϵ, signifying an empty string. Every path from *trie.root* to a terminal node corresponds to a token sequence from $\mathbb{C}$. As an illustration, Figure 5 presents the prefix tree corresponding to the 68 completion suggestions shown in Figure 1. In this example, nodes enclosed in blue boxes indicate the terminals of token sequences.

Subsequently, the algorithm proceeds to select a path in *trie* in a greedy fashion, based on predictions made by the fine-tuned code LLM, and appends the token sequence associated with the chosen path to the incomplete

Algorithm 3: Suggestion Selection based on Constraint Greedy Search

1 **Procedure** LLM-SELECT($\mathcal{D}, \mathcal{F}, \mathbb{C}$):
2 $\quad trie \leftarrow Trie()$ // prefix tree
3 $\quad$ **for** $comp$ **in** $\mathbb{C}$ **do**
4 $\quad\quad seq \leftarrow$ LLM-TOKENIZE$(comp)$
5 $\quad\quad trie.insert(seq)$
6 $\quad node \leftarrow trie.root$
7 $\quad$ **while not** $node.is\_terminal$ **do**
8 $\quad\quad \boldsymbol{m}^{|\mathbb{V}|} \leftarrow \boldsymbol{0}$
9 $\quad\quad$ **for** $child$ **in** $node.children$ **do**
10 $\quad\quad\quad \boldsymbol{m}^{|\mathbb{V}|}[child.tok\_idx] \leftarrow 1$
11 $\quad\quad \boldsymbol{p}^{|\mathbb{V}|} \leftarrow$ LLM-PREDICT$(\mathcal{D}, \mathcal{F})$
12 $\quad\quad \boldsymbol{p}^{|\mathbb{V}|} \leftarrow \boldsymbol{p}^{|\mathbb{V}|} \odot \boldsymbol{m}^{|\mathbb{V}|}$
13 $\quad\quad tok \leftarrow$ ARGMAX$(\mathbb{V}, \boldsymbol{p}^{|\mathbb{V}|})$
14 $\quad\quad \mathcal{F} \leftarrow \mathcal{F} \oplus tok$
15 $\quad\quad$ **for** $child$ **in** $node.children$ **do**
16 $\quad\quad\quad$ **if** $child.token = tok$ **then**
17 $\quad\quad\quad\quad node \leftarrow child$
18 $\quad\quad\quad\quad$ **break**

function $\mathcal{F}$ (lines 6-13). Specifically, the algorithm initiates a node pointer, denoted as *node*, with the root node *trie.root* (line 6). A loop continues until the pointer *node* reaches a terminal node (line 7). Within this loop, a $|\mathbb{V}|$-dimensional mask vector, denoted as $\boldsymbol{m}^{|\mathbb{V}|}$, is generated based on the children of the current node (lines 8-10). In $\boldsymbol{m}^{|\mathbb{V}|}$, only positions corresponding to the *tok_idx* property of the children of *node* are assigned a value of 1, while all other positions are set to 0. Subsequently, the fine-tuned code LLM is employed to predict a probability distribution, $\boldsymbol{p}^{|\mathbb{V}|}$ (line 11). This predicted distribution is then element-wise multiplied by the mask vector $\boldsymbol{m}^{|\mathbb{V}|}$, effectively setting the probability of tokens not in the children of *node* to 0. The next token, *tok*, is selected from $\mathbb{V}$ based on the highest probability in $\boldsymbol{p}^{|\mathbb{V}|}$ using the ARGMAX function and is appended to the current incomplete function $\mathcal{F}$ (lines 13-14). Finally, the node pointer is updated to point to the child of *node* whose stored token matches the selected token *tok* (line 15-18).

Example: For the prefix tree illustrated in Figure 5, the LLM-SELECT procedure iteratively selects the next tokens within the tree, guided by the LLM's predictions. This iterative process results in the inclusion of tokens corresponding to the suggestion "`_registered_updates`", which are found along the **green path**, being appended to the incomplete function.

4 EVALUATION SETUP

To evaluate the effectiveness and efficiency of ToolGen in repository-level code generation, we conduct a comprehensive set of experiments.

4.1 Research Questions

We formulate the following research questions to guide our evaluation:

- **RQ1 - Similarity-based Effectiveness:** How closely does the code generated by ToolGen align with the ground truth when assessed using common similarity metrics?
- **RQ2 - Dependency-based Effectiveness:** To what degree can ToolGen cover repository-level dependencies and reduce dependency errors, including those related to user-defined functions and attributes?
- **RQ3 - Execution-based Effectiveness:** How effectively can ToolGen generate functionally correct functions that pass test cases?
- **RQ4 - Efficiency:** What is the average time ToolGen takes to generate functions?
- **RQ5 - Generalizability:** Is ToolGen effective in code generation when applied to different code LLMs?

4.2 Implementation

Although ToolGen is designed to be language-agnostic, our current focus is on developing a Python-specific prototype of ToolGen.

Base Model. In ToolGen, we explore the utilization of three distinct code LLMs to encompass diverse model architectures and parameter scales. These code LLMs demonstrate impressive performance in code generation and have found extensive utilization in prior studies [31, 37, 51, 53, 54] for fine-tuning and evaluation.

- **CodeGPT:** CodeGPT [31] falls into the category of decoder-only models. It undergoes pre-training on a Python corpus sourced from the CodeSearchNet dataset [25], comprising 1.1 million Python functions. For our purposes, we adopt the pre-trained *CodeGPT-small* version[1], which encompasses 124 million model parameters.
- **CodeT5:** CodeT5 [54] belongs to the encoder-decoder model category and is similarly pre-trained on the Python corpus from the CodeSearchNet dataset. We select the pre-trained *CodeT5-base* version[2], which comprises 220 million model parameters.
- **CodeLlama:** CodeLlama [37] represents another decoder-only model, specialized for code-related tasks and based on Llama2 [43]. It is pre-trained on an even larger Python corpus, encompassing a staggering 100 billion tokens sourced from a Python-centric dataset [37]. For our purposes, we adopt the pre-trained *CodeLlama-7b* version[3], featuring a substantial 7 billion model parameters.

We refer to the variants of ToolGen, namely **ToolGen-*gpt***, **ToolGen-*t5***, and **ToolGen-*llama***, corresponding to the underlying base models CodeGPT, CodeT5, and CodeLlama, respectively.

Autocompletion Tool. We employ Jedi [2] as our autocompletion tool. Jedi is a static analysis tool designed for Python, commonly utilized within integrated development environments (IDEs) and editor plugins. Utilizing Jedi, ToolGen can trigger autocompletion, generating a list of suggestions that encompasses *repository-level dependencies*, including user-defined attributes and functions.

Trigger Insertion. To create the augmented dataset for fine-tuning the employed base model, we begin with the Python corpus from the *training set* of CodeSearchNet dataset. Since the CodeSearchNet dataset does not provide complete code repositories from which to extract Python functions, we initiate the process by crawling the code repositories listed in the dataset. Subsequently, we follow the procedure outlined in Section 3.2.1 to extract and augment functions within these code repositories, ultimately generating the augmented dataset. It's important to note that the CodeSearchNet dataset includes a partitioning into training, validation, and test sets. For our trigger insertion process, we exclusively utilize the code repositories associated with the training set. The resulting augmented dataset comprises a total of 249,298 pairs of descriptions and augmented functions, which are sourced from 12,231 distinct Python code repositories. Regarding dataset statistics, the average token count

[1]https://huggingface.co/microsoft/CodeGPT-small-py
[2]https://huggingface.co/Salesforce/codet5-base
[3]https://huggingface.co/codellama/CodeLlama-7b-Python-hf

for descriptions is 10.98, and for augmented functions, it is 55.31. Additionally, the special token `<COMP>` appears an average of 5.54 times within these functions.

Model Fine-tuning. In the fine-tuning process, we adopt different strategies for CodeGPT, CodeT5, and CodeLlama: For CodeGPT and CodeT5, we perform full-parameter fine-tuning, optimizing all model parameters during this phase. In the case of CodeLlama, we employ LoRA with a reduction factor (r) of 8 and a scaling factor ($alpha$) of 16 to achieve parameter-efficient fine-tuning. This approach allows us to optimize only 3.86% of the trainable parameters in comparison to the original CodeLlama model. The fine-tuning settings for learning rate and batch size are consistent across all three models, with a learning rate of 5E-6 and a batch size of 32. However, the number of epochs differs: 10 epochs for CodeGPT and CodeT5, while CodeLlama undergoes fine-tuning for 3 epochs. To ensure reproducibility, we set the seed for random functions to 42 consistently across all packages and libraries used.

4.3 Evaluation Benchmark

4.3.1 Datasets. To evaluate ToolGen, we curate two datasets: (i) a large dataset derived from the CodeSearchNet [25] to assess similarity-based and dependency-based effectiveness (RQ1 and RQ2); (ii) a dataset derived from CoderEval [56] containing test cases to evaluate execution-based effectiveness (RQ3).

CodeSearchNet. To assess similarity-based and dependency-based effectiveness, we follow this process to construct the dataset: We start by crawling the code repositories listed in the *test set* of the CodeSearchNet dataset, ensuring no overlap with the *training set* used for model fine-tuning. We then extract pairs of descriptions and functions from these repositories by parsing and traversing Abstract Syntax Trees (ASTs), similar to the method described in Section 3.2.1. This process yields an *evaluation dataset* comprising 12,406 Python functions sourced from 671 code repositories. On average, the descriptions contain 10.66 tokens, while the functions consist of an average of 54.54 tokens.

CoderEval. To evaluate execution-based effectiveness, we initially gather all 230 Python code generation tasks from the CoderEval benchmark, extracted from 43 real-world Python repositories. Each task consists of a natural language description, a ground-truth code snippet, and a set of test cases, along with the project environment context associated with the task (e.g., project source code, dependent libraries, and test scripts). The tasks are categorized into six runnable levels: self-contained, slib-runnable, plib-runnable, class-runnable, file-runnable, and project-runnable [56]. Each runnable level relies on the dependencies defined at that level and does not depend on those defined at subsequent levels. For example, plib-runnable indicate that the task requires public third-party libraries, while file-runnable require dependencies defined in the current file (e.g., user-defined classes and functions). We remove the tasks overlapping with the training dataset of ToolGen, resulting in a final dataset containing 176 tasks.

4.3.2 Baselines. The different variants of ToolGen and the baselines are presented in Table 1, along with the base models they employ.

Vanilla Baselines. We develop three vanilla baseline approaches by fine-tuning these same base models but performing straightforward code generation without tool integration. Specifically, the fine-tuning process for the baselines involves using the 249,298 pairs of descriptions and functions from the *augmented dataset*. Notably, the fine-tuning is conducted on the original functions, prior to the introduction of `<COMP>` tokens. The training configurations, including learning rates and training epochs, mirror those employed in the implementation of ToolGen. After fine-tuning, these models are utilized to perform straightforward code generation, as outlined in Section 2.1. We label the three baseline approaches as follows:

- **Vanilla-*gpt***: Represents straightforward code generation using the CodeGPT model fine-tuned on original functions.

Table 1. Variants of ToolGen and baselines.

Approach	Base Model	Architecture	# Parameters
Vanilla-*gpt*	CodeGPT	Decoder-Only	124 Million
RepoCoder-*gpt*			
ToolGen-*gpt* (ours)			
*rag*ToolGen-*gpt* (ours)			
Vanilla-*t5*	CodeT5	Encoder-Decoder	220 Million
RepoCoder-*t5*			
ToolGen-*t5* (ours)			
*rag*ToolGen-*t5* (ours)			
Vanilla-*llama*	CodeLlama	Decoder-Only	7 Billion
RepoCoder-*llama*			
ToolGen-*llama* (ours)			
*rag*ToolGen-*llama* (ours)			

- **Vanilla-*t5***: Signifies straightforward code generation using the CodeT5 model fine-tuned on original functions.
- **Vanilla-*llama***: Designates straightforward code generation with the CodeLlama model fine-tuned on original functions.

Retrieval-Augmented-Generation (RAG) Baselines. We also include RepoCoder [58], a state-of-the-art approach that addresses repository-level code generation by integrating a similarity-based retriever and a pre-trained code language model in an iterative retrieval-augmented-generation pipeline. Similarly, we create three variants of RepoCoder based on the three fine-tuned models Vanilla-*gpt*, Vanilla-*t5*, and Vanilla-*llama*. We directly apply the prompt template defined in the original implementation of RepoCoder. The three variants of RepoCoder are listed as follows:

- **RepoCoder-*gpt***: Represents the variant of RepoCoder with the CodeGPT model fine-tuned on original functions.
- **RepoCoder-*t5***: Signifies the variant of RepoCoder with the CodeT5 model fine-tuned on original functions.
- **RepoCoder-*llama***: Designates the variant of RepoCoder with the CodeLlama model fine-tuned on original functions.

The hyperparameters of RepoCoder used in our experiments follow its default implementation: the retrieval-generation iteration is set to 2, the window size is 20, and the slice size is 2.

RAG-based Variants of ToolGen. In fact, RAG method is orthogonal to our tool-integrated approach. To ensure a fair comparison and further explore the potential of ToolGen, we also develop three RAG-based variants: ***rag*ToolGen-*gpt***, ***rag*ToolGen-*t5***, and ***rag*ToolGen-*llama***. In these variants, the employed retrieval process is the same as the RepoCoder baselines.

4.3.3 Metrics. In our evaluation, we employ commonly used similarity-based metrics, two novel dependency-based metrics, and an execution-based metric to evaluate the effectiveness of ToolGen in repository-level code generation.

Similarity-based Metrics: We utilize the following well-established similarity metrics to measure the correspondence between generated functions and their ground-truth counterparts:

- **BLEU-4** [34]: This metric assesses the quality of generated code by comparing n-grams (sequences of n consecutive tokens) in the generated functions with those in the ground-truth functions.
- **CodeBLEU** [31]: Specifically designed for code generation tasks, CodeBLEU evaluates the accuracy of code generation models by considering code-specific vocabulary and structure.
- **Edit Similarity (EditSim)** [41]: This metric measures the similarity between two pieces of functions by analyzing the character-level edit operations required to transform one into the other.
- **Exact Match**: This metric measures the ratio of the generated code that are exactly matched with the ground truth.

The calculation of the similarity-based metrics follows the implementation in CodeXGLUE[4].

Dependency-based Metrics: To assess the effectiveness of both ToolGen and the baselines in repository-level code generation, we introduce the dependency-based metrics, namely *Dependency Coverage* (DepCov) and *Static Validity Rate* (ErrRate).

- ***Dependency Coverage*** **(DepCov):** This metric calculates the ratio of repository-level dependencies, including user-defined functions and attributes, that appear in ground-truth functions and are covered by the generated functions. Given the i-th ground-truth function gt_i, we identify dependencies by performing the Trigger Insertion procedure (Algorithm 1) and extracting expressions (such as function calls and attribute accesses like "`self._registered_updates`") that are marked with a trigger. These expressions are considered dependencies as their definitions can be traced in the current repository using static analysis tools like Jedi. Next, for the generated function $pred_i$ corresponding to gt_i, we extract all expressions by traversing its corresponding AST. We denote the identified dependencies in gt_i and extracted expressions in $pred_i$ as two sets, $DEP(gt_i)$ and $EXP(pred_i)$, respectively. The *Dependency Coverage* can be calculated as follows, where N is the size of the test dataset:

$$DepCov = \frac{\sum_i^N |EXP(pred_i) \cap DEP(gt_i)|}{\sum_i^N |DEP(gt_i)|}$$

- ***Static Validity Rate*** **(ValRate):** As repository-level dependencies can potentially introduce dependency errors in generated functions, we introduce the *Static Validity Rate* metric (ValRate) to evaluate the effectiveness of ToolGen in reducing dependency errors. This metric evaluates the proportion of generated functions that successfully pass a static check for dependency errors, specifically *no-member* and *undefined-variable*. To perform this evaluation, we incorporate the generated functions into their respective code repositories and conduct static lint analysis using pylint [3]. Functions that do not exhibit syntax errors, *no-member*, or *undefined-variable* errors are deemed statically valid. The *Static Validity Rate* can be calculated as follows:

$$ValRate = \frac{|\{pred_{i \le N} : pred_i \text{ passes lint check}\}|}{N}$$

Execution-based Metric: To further assess the functional correctness of the generated functions, we also employ a widely used execution-based metric involving running test cases.

[4]https://github.com/microsoft/CodeXGLUE

Table 2. Evaluation results of similarity-based effectiveness. Δ indicates the metric improvement or reduction of ToolGen's variants compared to the baselines.

Approach	**Similarity-based Metrics**			
	BLEU-4	CodeBLEU	EditSim	ExactMatch
Vanilla-*gpt*	0.331	0.313	65.4%	4.2%
ToolGen-*gpt*	0.340 (Δ = +2.7%)	0.310 (Δ = −1.0%)	64.7% (Δ = −1.1%)	4.7% (Δ = +11.9%)
Vanilla-*t5*	0.341	0.289	63.9%	4.3%
ToolGen-*t5*	0.362 (Δ = +6.2%)	0.293 (Δ = +1.4%)	61.9% (Δ = −3.1%)	5.5% (Δ = +27.9%)
Vanilla-*llama*	0.408	0.360	67.9%	5.7%
ToolGen-*llama*	0.425 (Δ = +4.2%)	0.358 (Δ = −0.6%)	66.3% (Δ = −2.4%)	6.9% (Δ = +21.1%)

- **Test Pass Rate (Pass@1):** This metric calculates the ratio of generated functionally-correct functions that pass all corresponding test cases. It is evaluated specifically on the CoderEval dataset, where test cases and test scripts are provided.

5 RESULTS AND ANALYSES

5.1 RQ1: Similarity-based Effectiveness

The evaluation results of similarity-based metrics are presented in Table 2. When comparing the performance of ToolGen's variants and the three different base models, namely CodeGPT, CodeT5, and CodeLlama, we observe that ToolGen achieves similarity scores comparable to the baselines.

To provide a detailed breakdown, when utilizing CodeGPT as the base model, ToolGen-*gpt* demonstrates a 2.7% improvement in BLEU-4 and a 11.9% improvement in Exact Match compared to Vanilla-*gpt*. However, it exhibits a 1.0% decrease in CodeBLEU and a 1.1% decrease in Edit Similarity. With the base model CodeT5, ToolGen-*t5* exhibits 6.2%, 1.4%, and 27.9% enhancements in BLEU-4, CodeBLEU, and Exact Match relative to Vanilla-*t5* but experiences a 3.1% decrease in Edit Similarity. In the case of the larger base model CodeLlama, ToolGen-*llama* shows improvements of 4.2% and 21.1% in BLEU-4 and Exact Match compared to Vanilla-*t5* but encounters a 0.6% decrease in CodeBLEU and a 2.4% decrease in Edit Similarity.

Although the absolute improvements in Exact Match rate are not large (from 0.5% to 1.2%), considering the size of the test set (*e.g.,* 12,406 samples), the additional exactly matched functions range from 62 to 149. The variability in ToolGen's performance across BLEU-4 and CodeBLEU can be attributed to the tokenization methods used in the calculations. For BLEU-4, before using the widely used utility script[5], we first tokenize the generated function and its corresponding ground-truth function using the tokenizer of the base code LLMs. In contrast, CodeBLEU is calculated based on the original generated and ground-truth code using the utility script[6] that employs a simpler method, splitting functions into strings (*e.g.,* "`func(arg1,)`") based on spaces. This splitting method may introduce inaccuracies in the statistics of matched n-grams, consequently affecting the CodeBLEU calculation. For Edit Similarity, it is calculated at the character level, making it overly sensitive to semantics-insensitive

[5]https://github.com/microsoft/CodeXGLUE/blob/main/Text-Code/text-to-code/evaluator/bleu.py
[6]https://github.com/microsoft/CodeXGLUE/tree/main/Code-Code/code-to-code-trans/evaluator/CodeBLEU

Table 3. Evaluation results of dependency-based effectiveness. DepCov and ValRate represent *Dependency Coverage* and *Static Validity Rate*, respectively. ValRate-*dep* represents the *Static Validity Rate* rate calculated only for functions containing dependencies. Δ indicates the metric improvement or reduction of ToolGen's variants compared to the baselines.

Approach	Dependency-based Metrics		
	DepCov	ValRate	ValRate-*dep*
Vanilla-*gpt*	8.7%	50.4%	46.5%
ToolGen-*gpt*	12.1% (Δ = +39.1%)	79.5% (Δ = +57.7%)	78.0% (Δ = +67.7%)
Vanilla-*t5*	11.0%	47.3%	42.5%
ToolGen-*t5*	15.0% (Δ = +36.4%)	70.6% (Δ = +49.3%)	68.0% (Δ = +60.0%)
Vanilla-*llama*	14.0%	49.7%	44.4%
ToolGen-*llama*	18.4% (Δ = +31.4%)	72.0% (Δ = +44.9%)	69.6% (Δ = +56.8%)

elements like temporary variables. When two variables have different names, their similarity is much lower at the character level than at the token level.

SUMMARY: ToolGen demonstrates competitive performance in similarity metrics compared to the baselines across various base models. It achieves improvements in BLEU-4 and Exact Match while exhibiting comparable performance in CodeBLEU and Edit Similarity.

5.2 RQ2: Dependency-based Effectiveness

5.2.1 Dependency Coverage. Table 3 displays the evaluation results for repository-level *Dependency Coverage* (DepCov). Notably, our approach ToolGen demonstrates significant superiority over the baselines across all three base models ($p \ll 0.01$). Specifically, when employing the base models CodeGPT, CodeT5, and CodeLlama, ToolGen surpasses the corresponding baselines in *Dependency Coverage* by 39.1%, 36.4%, and 31.4%, respectively.

These results underscore the effectiveness of the tool-integrated generation process in enhancing awareness of repository-level dependencies, a challenge often unaddressed by the conventional code LLMs. For instance, consider the incomplete function in Figure 1: in a straightforward CodeLlama generation, it fails to recognize the valid attributes of "`self`". However, through tool-integrated generation, ToolGen leverages Jedi to deduce a list of completion suggestions, enabling it to select the most suitable one and cover target repository-level dependencies, including the usage of user-defined functions and attributes.

Despite the considerable improvement in repository-level *Dependency Coverage* facilitated by our approach, it is essential to acknowledge that the overall coverage remains limited. This limitation arises from the fact that code LLMs generate function tokens sequentially from left to right. Consequently, errors tend to accumulate as the token count increases due to the exposure bias problem [8, 11, 36]. This means that code LLMs often make incorrect token predictions at certain generation steps and may not produce `<COMP>` tokens to trigger autocompletion tools, especially for long functions.

SUMMARY: Our approach, ToolGen, consistently outperforms the baselines in repository-level *Dependency Coverage* across all three base models by ranging from 31.4% to 39.1%. These results highlight the effectiveness of our tool-integrated generation process in addressing the crucial issue of enhancing awareness of repository-level dependencies, which is often a challenge for conventional code LLMs in repository-level code generation.

5.2.2 Static Validity Rate. Table 3 also presents the evaluation results for *Static Validity Rate* (ValRate and ValRate-*dep*) in repository-level lint analysis, with a particular focus on *no-member* and *undefined-variable* errors. Remarkably, our approach, ToolGen, consistently exhibits significantly higher *Static Validity Rate* compared to the baselines across all three base models ($p \ll 0.01$). Specifically, when employing the base models CodeGPT, CodeT5, and CodeLlama, ToolGen increases the *Static Validity Rate* (ValRate) by 57.7%, 49.3%, and 44.9%, respectively. When considering only the functions containing dependencies, ToolGen improves the *Static Validity Rate* (ValRate-*dep*) by 67.7%, 60.0%, and 56.8%, respectively.

These results underscore the effectiveness of our tool-integrated generation process in mitigating the production of invalid identifiers during code generation within a specific repository context. For instance, let's revisit the incomplete function in Figure 1: in a straightforward CodeLlama generation, it may predict a non-existent attribute, such as "`updates`", for "`self`". In contrast, through our tool-integrated approach, only valid completion suggestions inferred by Jedi are considered as candidates, thereby preventing numerous *no-member* and *undefined-variable* errors.

SUMMARY: Our approach, ToolGen, consistently achieves significantly higher *Static Validity Rate* in repository-level lint analysis compared to the baselines, with improvements ranging from 44.9% to 57.7%. These results underscore the effectiveness of our tool-integrated generation process in mitigating the generation of invalid identifiers, a common challenge faced by conventional code LLMs in the context of repository-level code generation.

5.3 RQ3: Execution-based Effectiveness

Table 4 presents the detailed evaluation results for test case execution (Pass@1) on the 176 CoderEval coding tasks.

Comparison to Vanilla baselines. Compared to the three Vanilla baselines, our approach ToolGen generates 0, 2, and 3 additional functionally-correct functions, resulting in 0%, 40.0%, and 25.0% improvements in Pass@1, respectively. Specifically, ToolGen-*gpt* improves the pass rate for *file-runnable* tasks, while reducing pass rate for *self-contained* tasks; ToolGen-*t5* improves pass rate for *slib-runnable file-runnable* tasks; ToolGen-*llama* improves pass rate for *self-contained*, *plib-runnable*, and *class-runnable* tasks. These tasks require different runnable-level dependencies (such as local variables and user-defined functions) to achieve correct functionality in the code. The enhancements by ToolGen underscore the effectiveness of integrating autocompletion tools to handle these dependencies.

Comparison to RepoCoder baselines. Overall, the RepoCoder baselines show unstable performance across different base models. Specifically, compared to their respective Vanilla baselines, RepoCoder-*gpt* and RepoCoder-*t5* exhibit reductions or no improvement in test pass rates, while RepoCoder-*llama* shows significant improvement. When compared to RepoCoder-*gpt* and RepoCoder-*t5*, our ToolGen-*gpt* and ToolGen-*t5* show improvements in pass rates with 1 and 2 more functions passing the test cases, respectively. However, ToolGen-*llama* exhibits a lower pass rate than RepoCoder-*llama* (15 vs. 19). These variations can be attributed to several factors: CodeGPT and CodeT5 have fewer parameters (124 million and 220 million) and stricter token number limitations (1,024 and 512), limiting their ability to process and understand retrieval-augmented prompts. In contrast, CodeLlama, with more model parameters (7 billion) and support for longer token sequences (16,384

Table 4. Evaluation results of execution-based effectiveness. *self*, *slib*, *plib*, *class*, *file*, and *project* represent self-contained, slib-runnable, plib-runnable, class-runnable, file-runnable, and project-runnable, respectively. The numbers in brackets after each runnable level indicate the corresponding number of tasks, while the numbers in brackets after the rates indicate the number of generated functions that pass the test cases. In each base model group, the best results are highlighted in gray, except when all results are the same.

Approach	**Execution-based Metric (Pass@1)**						
	total (176)	*self* (26)	*slib* (23)	*plib* (15)	*class* (49)	*file* (51)	*project* (12)
Vanilla-*gpt*	3.4% (6)	7.7% (2)	4.3% (1)	6.7% (1)	2.0% (1)	2.0% (1)	0.0% (0)
ToolGen-*gpt* (ours)	3.4% (6)	3.8% (1)	4.3% (1)	6.7% (1)	2.0% (1)	3.9% (2)	0.0% (0)
RepoCoder-*gpt*	2.8% (5)	0.0% (0)	4.3% (1)	6.7% (1)	4.1% (2)	2.0% (1)	0.0% (0)
*rag*ToolGen-*gpt* (ours)	2.8% (5)	0.0% (0)	4.3% (1)	6.7% (1)	4.1% (2)	2.0% (1)	0.0% (0)
Vanilla-*t5*	4.0% (7)	7.7% (2)	4.3% (1)	13.3% (2)	4.1% (2)	0.0% (0)	0.0% (0)
ToolGen-*t5* (ours)	5.1% (9)	15.4% (4)	8.7% (2)	6.7% (1)	4.1% (2)	0.0% (0)	0.0% (0)
RepoCoder-*t5*	4.0% (7)	7.7% (2)	4.3% (1)	13.3% (2)	4.1% (2)	0.0% (0)	0.0% (0)
*rag*ToolGen-*t5* (ours)	5.1% (9)	15.4% (4)	8.7% (2)	6.7% (1)	4.1% (2)	0.0% (0)	0.0% (0)
Vanilla-*llama*	6.8% (12)	19.2% (5)	13.0% (3)	13.3% (2)	4.1% (2)	0.0% (0)	0.0% (0)
ToolGen-*llama* (ours)	8.5% (15)	23.1% (6)	13.0% (3)	20.0% (3)	4.1% (2)	2.0% (1)	0.0% (0)
RepoCoder-*llama*	10.8% (19)	26.9% (7)	17.4% (4)	0.0% (0)	10.2% (5)	2.0% (1)	16.7% (2)
*rag*ToolGen-*llama* (ours)	10.8% (19)	34.6% (9)	8.7% (2)	6.7% (1)	6.1% (3)	0.0% (0)	33.3% (4)

tokens), allows RepoCoder-*llama* to achieve a higher pass rate than Vanilla-*llama* and ToolGen-*llama* due to the benefits of RAG.

Integration with RAG. When integrating ToolGen with RAG, *rag*ToolGen-*gpt* and *rag*ToolGen-*t5* do not show improvement, while *rag*ToolGen-*llama* exhibits breakthroughs for *project-runnable* tasks, with 4 more generated functions passing the test cases. However, the overall pass rate remains unchanged after integrating RAG, showing different advantages and disadvantages of RAG integration for various runnable-level dependencies.

SUMMARY: Our approach, ToolGen, outperforms or matches the three Vanilla baselines (Vanilla-*gpt*, Vanilla-*t5*, and Vanilla-*llama*) by generating 0, 2, and 3 more functionally correct functions, achieving 0%, 40.0%, and 25.0% improvements in Pass@1, respectively. Compared to RAG-based RepoCoder baselines, both ToolGen and RepoCoder have their own advantages and disadvantages for different base models and runnable-level dependencies. Additionally, our decoding-stage tool integration approach shows potential when combined with prompt-level RAG techniques for addressing certain types of dependencies.

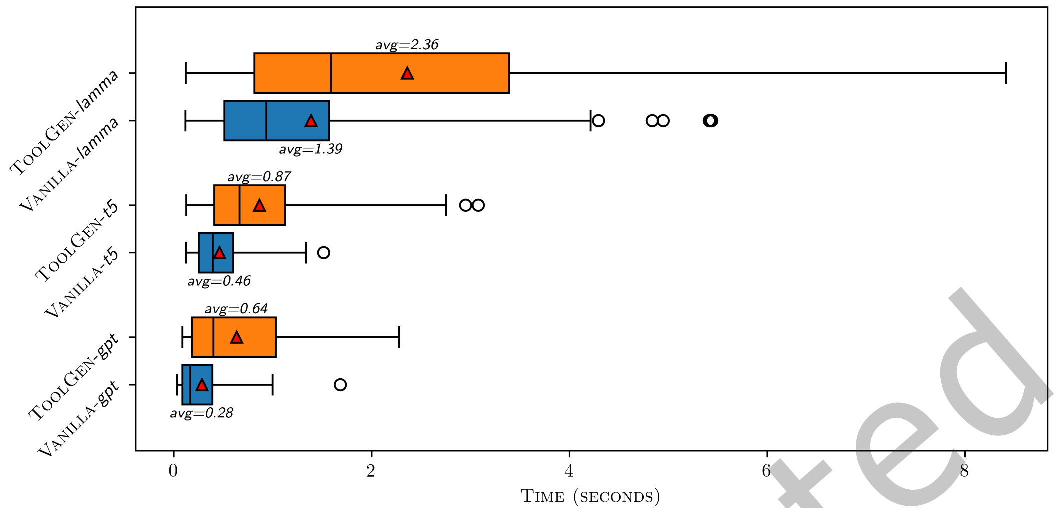


Fig. 6. Box Plot of Response Latency.

5.4 RQ4: Efficiency

Figure 6 illustrates the efficiency evaluation conducted on a single NVIDIA H100 Tensor Core GPU (80GB GPU Memory) without mini-batches (*i.e.,* batch size is 1). Our approach exhibits approximately twice the average generation time for the 176 tasks in the CoderEval dataset, showing 0.64, 0.87, and 2.36 seconds across the three base models. Note that using the same base models, RepoCoder-*gpt*, RepoCoder-*t5*, and RepoCoder-*llama* experience latencies of 0.80, 1.09, and 4.06 seconds, respectively. This indicates that RepoCoder incurs a higher latency overhead compared to ToolGen, as it significantly increases the number of input tokens, leading to substantially higher computational costs.

The high efficiency of our tool integration is attributed to the offline trigger insertion and fine-tuning. The autocompletion tool is triggered only when the fine-tuned models predict the trigger token `<COMP>`, significantly reducing unnecessary tool invocations. Specifically, the fine-tuned CodeGPT, CodeT5, and CodeLlama predict an average of 5.02, 6.24, and 7.05 `<COMP>` tokens per task, respectively, which is much fewer than the average function length. Additionally, during the generation of a function, autocompletion is often triggered multiple times for the same objects (*e.g.,* "`self`"); we maintain a cache to recall completion suggestions for previously visited objects, thereby avoiding repeated tool invocations for the same objects. While ToolGen shows improvements in effectiveness with only an acceptable increase in latency, further efficiency optimizations are necessary for more practical application. For example, implementing parallel background processes to inspect object creation during decoding and preemptively invoking autocompletion tools to cache potential candidates. When a `<COMP>` is predicted, the cached candidates can be retrieved instantly, reducing response time.

SUMMARY: Our tool-integrated generation approach, ToolGen, demonstrates high efficiency in repository-level code generation, with latency ranging from 0.63 to 2.34 seconds for generating each function. This efficiency is attributed to predicting the trigger token `<COMP>` and implementing a caching mechanism for completion suggestions.

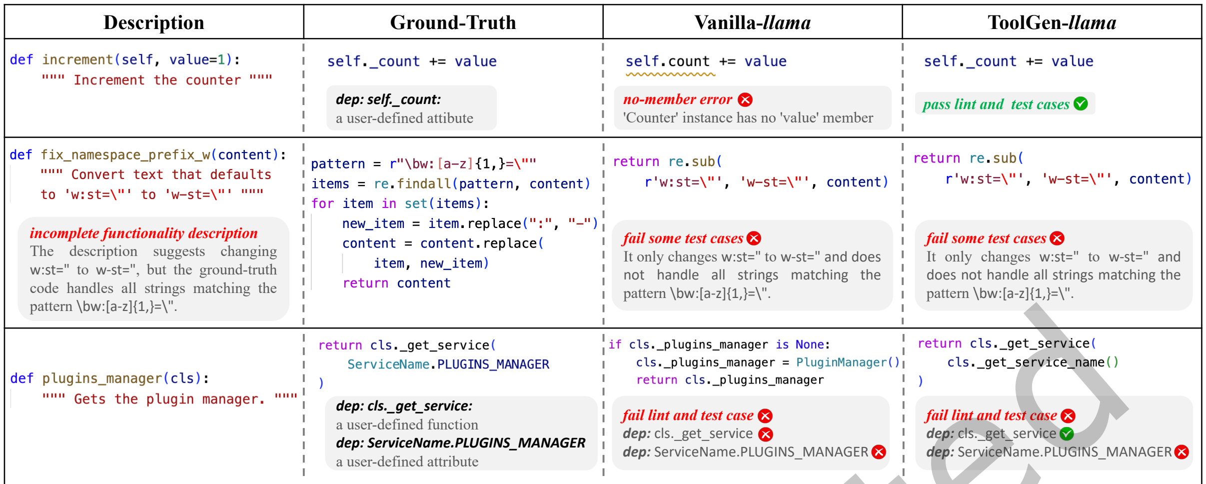


Fig. 7. Case study of three specific examples. Additional explanatory notes are marked with gray boxes. In the notes, "dep: xxx" denotes a dependency necessary in the generated code.

5.5 RQ5: Generalizability

Based on the results presented in Table 2 and Table 3, our tool-integrated generation approach consistently enhances performance in dependency-based metrics while maintaining comparable similarity-based metrics across various model architectures (decoder-only and encoder-decoder) and parameter scales (ranging from 124 million to 7 billion). According to Table 4 and Figure 6, our approach improves or maintains execution-based metrics across the base models, with a consistent and acceptable additional latency overhead.

SUMMARY: Our tool-integrated generation approach consistently improves or maintains dependency-based and execution-based metrics while achieving competitive similarity-based metrics across various model architectures and parameter scales. This suggests that our approach is versatile and has the potential for broader applicability with other base models in repository-level code generation.

6 DISCUSSION

6.1 Case Study

Figure 7 depicts three specific examples using Vanilla-*llama* and ToolGen-*llama*. Each row corresponds to an example, presenting the description, ground truth, code generated by Vanilla-*llama*, and code generated by ToolGen-*llama*.

Example 1: The code generated by ToolGen-*llama* successfully predicts the member "`_value`" in the class "`Counter`", while Vanilla-*llama* incorrectly predicts an undefined member "`value`", resulting in a *no-member* error. This difference can be attributed to ToolGen's integration of the autocompletion tool, which helps the code LLMs recognize necessary dependencies like user-defined attributes/members.

Example 2: Both Vanilla-*llama* and ToolGen-*llama* generate incorrect code that fails some test cases. After reviewing the description and the ground-truth, we find that the description is incomplete in expressing the desired functionality. As noted, the description only mentions changing "`w:st=''`" to "`w-st=''`", but the actual desired functionality in the ground-truth is to handle all strings matching the pattern "`\bw:[a-z]{1,}=\''`".

Both Vanilla-*llama* and ToolGen-*llama* follow the description and generate code that satisfies this incomplete functionality. This finding highlights the challenges posed by low-quality descriptions in real-world generation scenarios and reveals quality issues in existing benchmarks.

Example 3: There are two crucial dependencies, namely "`cls._get_service()`" and "`ServiceName.PLUGINS_MANAGER`", necessary to realize the required functionality. Vanilla-*llama* fails to predict both dependencies and instead outputs non-existent dependencies like "`cls._plugins_manager`" and "`PluginManager()`", causing the generated code to fail lint checks and test cases. For ToolGen-*llama*, although it successfully predicts the dependency "`cls._get_service()`", it fails to predict "`ServiceName.PLUGINS_MANAGER`" because the model chooses "`cls`" instead of "`ServiceName`" when starting predicting the argument for "`cls._get_service()`". This misleads the generation in an incorrect direction, resulting in the failure of the final code. This example also highlights the challenges of applying code LLMs in practical code generation, even when integrating autocompletion tools to avoid certain dependency issues. Introducing an incorrect token at any critical step in the generation process can result in the production of erroneous code.

6.2 Limitations

Static Autocompletion Tools for Dynamically Typed Programming Languages: Currently, our implementation of ToolGen is specific to Python, a dynamically typed programming language. However, the autocompletion tools used in ToolGen rely on static analysis, which can sometimes fail to trigger for certain repository-level dependencies. For instance, when the type of a function parameter cannot be explicitly inferred through static analysis, autocompletion tools may struggle to deduce attributes defined within the argument type. In the future, we plan to explore the integration of comprehensive type inference tools, such as learning-based tools, into the code generation process alongside autocompletion tools to enhance Python code generation.

Greedy Next Token Prediction in Generation Process: During the generation process, we employ a greedy strategy for next token prediction, where the token with the highest probability is selected using the argmax function. This greedy prediction strategy can occasionally lead the model to choose sub-optimal tokens for subsequent steps, resulting in code that may not be of the high quality. To address this issue, we intend to investigate the incorporation of techniques such as beam search and other advanced decoding methods into our tool-integrated generation process to mitigate the challenges posed by greedy prediction.

Dependency-based Evaluation Metrics: In the computation of the two repository-level evaluation metrics, namely *Dependency Coverage* and *Static Validity Rate*, we employ static analysis to identify target expressions and perform lint examinations. Similar to the autocompletion tools, these static tools may introduce a degree of inaccuracy into the calculated metrics. However, it is essential to note that this does not significantly impact the demonstrated effectiveness of ToolGen, as the baseline metrics are also determined using the same static analysis.

Integration and Comparison with SOTA Closed-source LLMs: Our approach can be applied to any encoder-decoder or decoder-only models. However, for the most state-of-the-art (SOTA) LLMs like GPT-3.5 and GPT-4, integrating tool-integrated decoding process faces challenges due to their closed-source nature. Although GPT-3.5 and GPT-4 can be fine-tuned remotely via OpenAI's fine-tuning platform[7], the process requires significant computational resources, and the models' internal decoding process cannot be modified or controlled. In the future, we may explore the possibility of integrating autocompletion tools into these closed-source LLMs through a fully prompt-based approach. In our evaluation, we do not compare ToolGen with these SOTA closed-source LLMs, as our goal is to assess the effectiveness of integrating autocompletion tools for repository-level code generation. Therefore, we focus on comparing the performance of ToolGen, Vanilla, and RepoCoder under the same base models.

[7]https://platform.openai.com/finetune

6.3 Threats to Validity

Internal Threats: The first internal threat pertains to potential data quality issues common in learning-based approaches. To mitigate this threat, we construct our augmented dataset and evaluation benchmark dataset using the widely adopted CodeSearchNet dataset, which serves as a reliable source for pretraining and evaluating various code models. Another internal threat pertains to the potential data leakage for CodeLlama, as the code repositories in the benchmark dataset may have been encountered by CodeLlama during its pretraining phase. However, our generalizability evaluation (RQ5) provides evidence of consistent performance across the three ToolGen variants, suggesting that the improvements achieved by ToolGen-*llama* in repository-level code generation are not attributed to data leakage.

External Threats: Our implementation and evaluation of ToolGen are specific to the Python programming language. As a result, the findings may not be generalizable to other programming languages. Exploring the tool-integrated generation process for different languages is a valuable direction for future research.

7 RELATED WORK

7.1 Code Generation

Code generation has long been a central focus in the field of software engineering. Recent developments have introduced a range of large language models for code (code LLMs), including Codex [15], CodeT5 [54], CodeT5+ [53], InCoder [21], AlphaCode [29], CodeGen [33], and CodeLlama [37], built upon the Transformer model architecture [45]. These models, either pretrained or fine-tuned on extensive code corpora, have the capability to automatically generate code based on provided natural language descriptions.

While these code LLMs have demonstrated significant effectiveness in generating standalone functions on existing benchmarks like HumanEval [15] and MBPP [9], they face substantial challenges when tasked with generating real-world functions within code repositories. The primary challenge stems from their lack of awareness of *repository-level dependencies*, such as user-defined functions and attributes, during the code generation process [56]. To address these challenges, researchers have proposed prompt engineering approaches to make code LLMs aware of repository-level dependencies. Shrivastava *et al.* [39] introduced the repository-level prompt generator, a framework for generating context-aware prompts without requiring access to the weights of the LLM. Bairi *et al.* [10] presented CodePlan, a task-agnostic framework that treats repository-level coding as a planning problem, using innovative techniques to generate multi-step code edits while considering context from the entire codebase, previous changes, and specific instructions.

In this study, we tackle the challenges associated with repository-level code generation by seamlessly integrating autocompletion tools into the generation process of code LLMs.

7.2 Incorporating External Tools into LLMs

Recent research [16, 18, 22, 26, 27, 32, 35, 38, 40, 42, 61] has explored the integration of external tools (*e.g.*, search engines, web browsers, calculators, and python interpreters) into the LLM generation process, aiming to address their limitations in certain generation scenarios. For instance, Schick *et al.* propose ToolFormer [38], which augments datasets to instruct LLMs on invoking existing arithmetic calculators, effectively reducing errors in generated text related to arithmetic calculations. Building upon this idea, Zhang *et al.* introduce ToolCoder [61], designed to teach LLMs how to utilize information-retrieval-based (IR-based) API search tools during code generation. While ToolCoder is effective in generating functionally correct standalone functions, it falls short in addressing *repository-level dependencies*, limiting its ability to resolve dependency errors. More relevant examples are Repilot [58], STALL+ [30], and MGD [6], which utilize code completion tools to filter out impractical suggestions made by LLMs, focusing on generating API/line-level code completions and valid bug-fix patches rather than entire functions.

In this paper, we integrate program-analysis-based autocompletion tools into the code LLM generation process to facilitate repository-level code generation.

8 CONCLUSION

We present ToolGen, an approach that seamlessly integrates autocompletion tools into the code LLM generation process to effectively address repository-level dependencies. ToolGen encompasses two crucial phases: Data Augmentation and Model Fine-tuning, and Tool-integrated Code Generation. Our comprehensive evaluation showcases ToolGen's improvements in the two introduced dependency-level metrics and a widely used execution-based metric across three distinct code LLMs, while also demonstrating its competitiveness in widely-recognized similarity metrics. ToolGen also demonstrates high efficiency in repository-level code generation, due to the offline fine-tuning with trigger insertion. Moreover, our generalizability evaluation reaffirms ToolGen's consistent performance when applied to diverse code LLMs, including various model architectures and scales.

9 DATA AVAILABILITY

All code and data can be found at our replication package [4].

ACKNOWLEDGEMENT

This research/project is supported by the National Key R&D Program of China (2023YFB4503805) and the National Research Foundation, Singapore, and the Cyber Security Agency under its National Cybersecurity R&D Programme (NCRP25-P04-TAICeN). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and Cyber Security Agency of Singapore.

REFERENCES

[1] 2024. *Argmax Function.* https://en.wikipedia.org/wiki/Arg_max
[2] 2024. *Jedi - an awesome autocompletion, static analysis and refactoring library for Python.* https://jedi.readthedocs.io/
[3] 2024. *Pylint.* https://github.com/pylint-dev/pylint
[4] 2024. *Replication Package.* https://github.com/cs-wangchong/ToolGen-Replication/
[5] 2024. *Trie Structure.* https://en.wikipedia.org/wiki/Trie
[6] Lakshya A Agrawal, Aditya Kanade, Navin Goyal, Shuvendu K Lahiri, and Sriram K Rajamani. 2023. Guiding language models of code with global context using monitors. *arXiv preprint arXiv:2306.10763* (2023).
[7] Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Muñoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, Logesh Kumar Umapathi, Carolyn Jane Anderson, Yangtian Zi, Joel Lamy-Poirier, Hailey Schoelkopf, Sergey Troshin, Dmitry Abulkhanov, Manuel Romero, Michael Lappert, Francesco De Toni, Bernardo García del Río, Qian Liu, Shamik Bose, Urvashi Bhattacharyya, Terry Yue Zhuo, Ian Yu, Paulo Villegas, Marco Zocca, Sourab Mangrulkar, David Lansky, Huu Nguyen, Danish Contractor, Luis Villa, Jia Li, Dzmitry Bahdanau, Yacine Jernite, Sean Hughes, Daniel Fried, Arjun Guha, Harm de Vries, and Leandro von Werra. 2023. SantaCoder: don't reach for the stars! *CoRR* abs/2301.03988 (2023). https://doi.org/10.48550/ARXIV.2301.03988 arXiv:2301.03988
[8] Kushal Arora, Layla El Asri, Hareesh Bahuleyan, and Jackie Chi Kit Cheung. 2022. Why Exposure Bias Matters: An Imitation Learning Perspective of Error Accumulation in Language Generation. In *Findings of the Association for Computational Linguistics: ACL 2022, Dublin, Ireland, May 22-27, 2022*, Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (Eds.). Association for Computational Linguistics, 700–710. https://doi.org/10.18653/V1/2022.FINDINGS-ACL.58
[9] Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *CoRR* abs/2108.07732 (2021). arXiv:2108.07732 https://arxiv.org/abs/2108.07732
[10] Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D. C, Arun Iyer, Suresh Parthasarathy, Sriram K. Rajamani, Balasubramanyan Ashok, and Shashank Shet. 2023. CodePlan: Repository-level Coding using LLMs and Planning. *CoRR* abs/2309.12499 (2023). https://doi.org/10.48550/ARXIV.2309.12499 arXiv:2309.12499
[11] Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. 2015. Scheduled Sampling for Sequence Prediction with Recurrent Neural Networks. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems*

2015, December 7-12, 2015, Montreal, Quebec, Canada, Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett (Eds.). 1171–1179. https://proceedings.neurips.cc/paper/2015/hash/e995f98d56967d946471af29d7bf99f1-Abstract.html

[12] Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, Michael Pieler, USVSN Sai Prashanth, Shivanshu Purohit, Laria Reynolds, Jonathan Tow, Ben Wang, and Samuel Weinbach. 2022. GPT-NeoX-20B: An Open-Source Autoregressive Language Model. *CoRR* abs/2204.06745 (2022). https://doi.org/10.48550/ARXIV.2204.06745 arXiv:2204.06745

[13] Sid Black, Leo Gao, Phil Wang, Connor Leahy, and Stella Biderman. 2021. *GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow.* https://doi.org/10.5281/zenodo.5297715 If you use this software, please cite it using these metadata..

[14] Huanchao Chen, Yuan Huang, Zhiyong Liu, Xiangping Chen, Fan Zhou, and Xiaonan Luo. 2019. Automatically detecting the scopes of source code comments. *J. Syst. Softw.* 153 (2019), 45–63. https://doi.org/10.1016/J.JSS.2019.03.010

[15] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

[16] Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. 2022. Program of Thoughts Prompting: Disentangling Computation from Reasoning for Numerical Reasoning Tasks. *CoRR* abs/2211.12588 (2022). https://doi.org/10.48550/ARXIV.2211.12588 arXiv:2211.12588

[17] Fenia Christopoulou, Gerasimos Lampouras, Milan Gritta, Guchun Zhang, Yinpeng Guo, Zhongqi Li, Qi Zhang, Meng Xiao, Bo Shen, Lin Li, Hao Yu, Li Yan, Pingyi Zhou, Xin Wang, Yuchi Ma, Ignacio Iacobacci, Yasheng Wang, Guangtai Liang, Jiansheng Wei, Xin Jiang, Qianxiang Wang, and Qun Liu. 2022. PanGu-Coder: Program Synthesis with Function-Level Language Modeling. *CoRR* abs/2207.11280 (2022). https://doi.org/10.48550/ARXIV.2207.11280 arXiv:2207.11280

[18] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training Verifiers to Solve Math Word Problems. *CoRR* abs/2110.14168 (2021). arXiv:2110.14168 https://arxiv.org/abs/2110.14168

[19] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2024. Evaluating Large Language Models in Class-Level Code Generation. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon , Portugal, April 14 - 20, 2024*. ACM, 1496–1508. https://doi.org/10.1145/3597503.3639219

[20] Xueying Du, Geng Zheng, Kaixin Wang, Jiayi Feng, Wentai Deng, Mingwei Liu, Bihuan Chen, Xin Peng, Tao Ma, and Yiling Lou. 2024. Vul-RAG: Enhancing LLM-based Vulnerability Detection via Knowledge-level RAG. *arXiv preprint arXiv:2406.11147* (2024).

[21] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. InCoder: A Generative Model for Code Infilling and Synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/pdf?id=hQwb-lbM6EL

[22] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. PAL: Program-aided Language Models. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 10764–10799. https://proceedings.mlr.press/v202/gao23f.html

[23] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-Rank Adaptation of Large Language Models. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net. https://openreview.net/forum?id=nZeVKeeFYf9

[24] Yuan Huang, Hanyang Guo, Xi Ding, Junhuai Shu, Xiangping Chen, Xiapu Luo, Zibin Zheng, and Xiaocong Zhou. 2023. A Comparative Study on Method Comment and Inline Comment. *ACM Trans. Softw. Eng. Methodol.* 32, 5 (2023), 126:1–126:26. https://doi.org/10.1145/3582570

[25] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *CoRR* abs/1909.09436 (2019). arXiv:1909.09436 http://arxiv.org/abs/1909.09436

[26] Mojtaba Komeili, Kurt Shuster, and Jason Weston. 2022. Internet-Augmented Dialogue Generation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (Eds.). Association for Computational Linguistics, 8460–8478. https://doi.org/10.18653/V1/2022.ACL-LONG.579

[27] Angeliki Lazaridou, Elena Gribovskaya, Wojciech Stokowiec, and Nikolai Grigorev. 2022. Internet-augmented language models through few-shot prompting for open-domain question answering. *CoRR* abs/2203.05115 (2022). https://doi.org/10.48550/ARXIV.2203.05115 arXiv:2203.05115

[28] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy V, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Moustafa-Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. StarCoder: may the source be with you! *CoRR* abs/2305.06161 (2023). https://doi.org/10.48550/ARXIV.2305.06161 arXiv:2305.06161

[29] Yujia Li, David H. Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-Level Code Generation with AlphaCode. *CoRR* abs/2203.07814 (2022). https://doi.org/10.48550/ARXIV.2203.07814 arXiv:2203.07814

[30] Junwei Liu, Yixuan Chen, Mingwei Liu, Xin Peng, and Yiling Lou. 2024. STALL+: Boosting LLM-based Repository-level Code Completion with Static Analysis. *arXiv preprint arXiv:2406.10018* (2024).

[31] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, Joaquin Vanschoren and Sai-Kit Yeung (Eds.). https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/c16a5320fa475530d9583c34fd356ef5-Abstract-round1.html

[32] Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, Xu Jiang, Karl Cobbe, Tyna Eloundou, Gretchen Krueger, Kevin Button, Matthew Knight, Benjamin Chess, and John Schulman. 2021. WebGPT: Browser-assisted question-answering with human feedback. *CoRR* abs/2112.09332 (2021). arXiv:2112.09332 https://arxiv.org/abs/2112.09332

[33] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/pdf?id=iaYcJKpY2B_

[34] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA*. ACL, 311–318. https://doi.org/10.3115/1073083.1073135

[35] Bhargavi Paranjape, Scott M. Lundberg, Sameer Singh, Hannaneh Hajishirzi, Luke Zettlemoyer, and Marco Túlio Ribeiro. 2023. ART: Automatic multi-step reasoning and tool-use for large language models. *CoRR* abs/2303.09014 (2023). https://doi.org/10.48550/ARXIV.2303.09014 arXiv:2303.09014

[36] Romain Paulus, Caiming Xiong, and Richard Socher. 2018. A Deep Reinforced Model for Abstractive Summarization. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=HkAClQgA-

[37] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. *CoRR* abs/2308.12950 (2023). https://doi.org/10.48550/ARXIV.2308.12950 arXiv:2308.12950

[38] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language Models Can Teach Themselves to Use Tools. *CoRR* abs/2302.04761 (2023). https://doi.org/10.48550/ARXIV.2302.04761 arXiv:2302.04761

[39] Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2023. Repository-Level Prompt Generation for Large Language Models of Code. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 31693–31715. https://proceedings.mlr.press/v202/shrivastava23a.html

[40] Kurt Shuster, Mojtaba Komeili, Leonard Adolphs, Stephen Roller, Arthur Szlam, and Jason Weston. 2022. Language Models that Seek for Knowledge: Modular Search & Generation for Dialogue and Prompt Completion. In *Findings of the Association for Computational Linguistics: EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang (Eds.). Association for Computational Linguistics, 373–393. https://doi.org/10.18653/V1/2022.FINDINGS-EMNLP.27

[41] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. IntelliCode compose: code generation using transformer. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 1433–1443. https://doi.org/10.1145/3368089.3417058

[42] Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, YaGuang Li, Hongrae Lee, Huaixiu Steven Zheng, Amin Ghafouri, Marcelo Menegali, Yanping Huang, Maxim Krikun, Dmitry Lepikhin, James Qin, Dehao Chen, Yuanzhong Xu, Zhifeng Chen, Adam Roberts, Maarten Bosma, Yanqi Zhou, Chung-Ching Chang, Igor Krivokon, Will Rusch, Marc Pickett, Kathleen S. Meier-Hellstern, Meredith Ringel Morris, Tulsee Doshi, Renelito Delos Santos, Toju Duke, Johnny Soraker, Ben Zevenbergen, Vinodkumar Prabhakaran, Mark Diaz, Ben Hutchinson, Kristen Olson, Alejandra Molina, Erin Hoffman-John, Josh Lee, Lora Aroyo, Ravi Rajakumar, Alena Butryna, Matthew Lamm, Viktoriya Kuzmina, Joe Fenton, Aaron Cohen, Rachel Bernstein, Ray Kurzweil, Blaise Agüera y Arcas, Claire Cui, Marian Croak, Ed H. Chi, and Quoc Le. 2022. LaMDA: Language Models for Dialog Applications. *CoRR* abs/2201.08239 (2022). arXiv:2201.08239 https://arxiv.org/abs/2201.08239

[43] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *CoRR* abs/2307.09288 (2023). https://doi.org/10.48550/ARXIV.2307.09288 arXiv:2307.09288

[44] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In *CHI '22: CHI Conference on Human Factors in Computing Systems, New Orleans, LA, USA, 29 April 2022 - 5 May 2022, Extended Abstracts*, Simone D. J. Barbosa, Cliff Lampe, Caroline Appert, and David A. Shamma (Eds.). ACM, 332:1–332:7. https://doi.org/10.1145/3491101.3519665

[45] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 5998–6008. https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[46] Ben Wang and Aran Komatsuzaki. 2021. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. https://github.com/kingoflolz/mesh-transformer-jax.

[47] Chong Wang, Kaifeng Huang, Jian Zhang, Yebo Feng, Lyuye Zhang, Yang Liu, and Xin Peng. 2024. How and Why LLMs Use Deprecated APIs in Code Completion? An Empirical Study. *arXiv preprint arXiv:2406.09834* (2024).

[48] Chong Wang, Jianan Liu, Xin Peng, Yang Liu, and Yiling Lou. 2023. Boosting Static Resource Leak Detection via LLM-based Resource-Oriented Intention Inference. *arXiv preprint arXiv:2311.04448* (2023).

[49] Chong Wang, Yiling Lou, Junwei Liu, and Xin Peng. 2023. Generating variable explanations via zero-shot prompt learning. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 748–760.

[50] Chong Wang, Jian Zhang, Yiling Lou, Mingwei Liu, Weisong Sun, Yang Liu, and Xin Peng. 2024. TIGER: A Generating-Then-Ranking Framework for Practical Python Type Inference. *arXiv preprint arXiv:2407.02095* (2024).

[51] Xin Wang, Yasheng Wang, Yao Wan, Fei Mi, Yitong Li, Pingyi Zhou, Jin Liu, Hao Wu, Xin Jiang, and Qun Liu. 2022. Compilable Neural Code Generation with Compiler Feedback. In *Findings of the Association for Computational Linguistics: ACL 2022, Dublin, Ireland, May 22-27, 2022*, Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (Eds.). Association for Computational Linguistics, 9–19. https://doi.org/10.18653/V1/2022.FINDINGS-ACL.2

[52] Yanlin Wang, Tianyue Jiang, Mingwei Liu, Jiachi Chen, and Zibin Zheng. 2024. Beyond Functional Correctness: Investigating Coding Style Inconsistencies in Large Language Models. *arXiv preprint arXiv:2407.00456* (2024).

[53] Yue Wang, Hung Le, Akhilesh Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, 1069–1088. https://aclanthology.org/2023.emnlp-main.68

[54] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, 8696–8708. https://doi.org/10.18653/V1/2021.EMNLP-MAIN.685

[55] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. 2023. Copiloting the Copilots: Fusing Large Language Models with Completion Engines for Automated Program Repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, Satish Chandra, Kelly Blincoe, and Paolo Tonella (Eds.). ACM, 172–184. https://doi.org/10.1145/3611643.3616271

[56] Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Tao Xie, and Qianxiang Wang. 2023. CoderEval: A Benchmark of Pragmatic Code Generation with Generative Pre-trained Models. *arXiv preprint arXiv:2302.00288* (2023).

[57] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. 2023. No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation. *CoRR* abs/2305.04207 (2023). https://doi.org/10.48550/ARXIV.2305.04207 arXiv:2305.04207

[58] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, 2471–2484. https://doi.org/10.18653/V1/2023.EMNLP-MAIN.151

[59] Junan Zhang, Kaifeng Huang, Bihuan Chen, Chong Wang, Zhenhao Tian, and Xin Peng. 2023. Malicious Package Detection in NPM and PyPI using a Single Model of Malicious Behavior Sequence. *arXiv preprint arXiv:2309.02637* (2023).

[60] Jian Zhang, Chong Wang, Anran Li, Weisong Sun, Cen Zhang, Wei Ma, and Yang Liu. 2024. An Empirical Study of Automated Vulnerability Localization with Large Language Models. *arXiv preprint arXiv:2404.00287* (2024).

[61] Kechi Zhang, Ge Li, Jia Li, Zhuo Li, and Zhi Jin. 2023. ToolCoder: Teach Code Generation Models to use APIs with search tools. *arXiv preprint arXiv:2305.04032* (2023).