Enhancing Domain-Specific Code Completion via Collaborative Inference with Large and Small Language Models

JINGRONG YU, Zhejiang University, China
ZHIPENG GAO, Zhejiang University, China
LINGFENG BAO, Zhejiang University, China
ZHONGXIN LIU*, Zhejiang University, China

Large language model-based code completion has demonstrated excellent performance, but still encounters challenges in capturing domain-specific knowledge for more precise completion within specific domains, i.e., domain-specific code completion. Prior work has studied fine-tuning techniques or retrieval-augmented techniques for this task. Nevertheless, it requires a lot of computational resources to fine-tune large language models (LLMs), and the cost can increase quadratically with the model size. Retrieval-augmented techniques face difficulties in accurately and adaptively retrieving relevant information. Moreover, considering that code completion tools work in real time, how to utilize large language models more efficiently poses challenges.

To tackle these challenges, in this paper, we first conduct preliminary experiments and observe that the code completion results of a small model fine-tuned within a specific domain complement those of a large model. Building on this insight, we propose a collaborative framework to effectively combine large and small models for better domain-specific code completion. Specifically, we fine-tune a small code model instead of a large model with the PEFT method, reducing the overhead of fine-tuning. We utilize a well-designed classifier to facilitate the adaptive combination of distinct completion results. The classifier relies on features in various dimensions, such as the similarity between the completed code and the context, and is used to adaptively determine how to combine the tokens predicted by the large and small models for better code completion. Evaluation results show that our approach achieves an average improvement in the exact match of 7.42% and 4.67% over the state-of-the-art baselines in the intra-project and intra-domain code completion scenarios, respectively. Furthermore, compared to the state-of-the-art domain-specific code completion approach FT2Ra, the inference speed of our approach is 1.40 times faster, and the average space requirement drops from 25.98G to 13.69G. These advantages make our approach much more accessible and efficient.

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

Additional Key Words and Phrases: Domain-Specific Code Completion, Large Language Model, Collaborative Inference

1 INTRODUCTION

Trained on extensive code sources, large language code models such as Deepseek-Coder [15], StarCoder [26, 32], CodeLlama [39], among others, have demonstrated strong capabilities and gained significant success in

*Corresponding author.

Authors' Contact Information: Jingrong Yu, yujingrong@zju.edu.cn, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China; Zhipeng Gao, zhipeng.gao@zju.edu.cn, Zhejiang University, China; Lingfeng Bao, lingfengbao@zju.edu.cn, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China; Zhongxin Liu, liu_zx@zju.edu.cn, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2025 Copyright held by the owner/author(s).
ACM 1557-7392/2025/10-ART
https://doi.org/10.1145/3770748

Just Accepted

code completion tasks. However, the performance of these code models notably lags when completing code within specific domains, i.e., domain-specific code completion, compared to general-purpose code completion scenarios [8], as large code models struggle to capture and learn complex code patterns and dependencies within specific domains [22].

Recent researches introduce fine-tuning [41, 58] and retrieval-augmented (RAG) techniques [17] to domain-specific code completion tasks. Fine-tuned with code within the corresponding domain, the code models can acquire sufficient domain-specific knowledge to perform the domain-specific code completion tasks effectively. However, the conventional full fine-tuning technique requires updating all model parameters, resulting in a substantial computational burden that scales quadratically with the size of code models [28]. RAG techniques aid in domain-specific code completion by retrieving information that potentially contributes to better completion. Retrieved information involving domain-specific knowledge is either provided as prompts to the code model, with the expectation that the model will utilize these prompts to improve code completion [8, 10, 43, 51] or combined with the code models' output to improve accuracy [16, 45]. However, RAG techniques face difficulties in accurately and adaptively retrieving relevant information to guide code models effectively. Furthermore, some existing RAG methods are sensitive to manually selected interpolation parameters [23] and require a large database for retrieval, incurring significant storage overhead and non-negligible retrieval costs [45]. Moreover, considering that code completion tools work in real time, how to utilize large language models more efficiently poses challenges.

To tackle the above challenges, in this paper, we propose a collaborative inference framework that simultaneously utilizes a large model and a fine-tuned small model for domain-specific code completion, incorporating the speculative decoding algorithm [25, 52]. We first conduct empirical experiments and observe that the outputs of a fine-tuned small model can effectively complement those of the large models. Building upon the finding, we leverage both a large model and a fine-tuned small model for domain-specific code completion. We fine-tune a relatively small code model with the PEFT method [35], to further mitigate the overhead of the traditional fine-tuning process of a large model. During code completion, we employ a well-designed classifier to effectively and adaptively combine the completion results from the large and small models, without manually selecting interpolated weights to combine the outputs of the models. The classifier relying on features in six dimensions, involving the probabilities and logits generated by the code models and retrieved information such as the occurrence frequencies and similarities of the currently completed code within the code context and corpus, helps combine the tokens predicted by the large and small models adaptively and effectively.

To demonstrate the effectiveness of our approach, we select the state-of-the-art domain-specific code completion methods kNM-LM [45] and FT2Ra [16] as baselines. We construct a new benchmark for intra-project and intra-domain line-level code completion, without using datasets from previous studies, tackling potential data leakage issues, as the large code models may already encounter the data during pre-training. The evaluation results on our benchmark demonstrate that our approach outperforms the baseline methods on intra-project and intra-domain code completion tasks. Specifically, our approach achieves an average exact match improvement of 7.42% and 4.67% compared to FT2Ra, and 9.13% and 5.57% compared to the large code model in intra-project and intra-domain code completion tasks, respectively. We further evaluate the time and space overhead of our approach and compare it with the SOTA method FT2Ra. The experiment results indicate that our approach achieves an approximately 1.67 times faster inference speed than FT2Ra, and also exhibits a substantial reduction from 24G to 1.3G of the space overhead. In addition, we perform an ablation study on the features selected for the classifier, revealing that all the features are rational and effective, contributing to the accuracy.

In summary, our main contributions are as follows:

Just Accepted

Table 1. Results of Line-level Code Completion on Exact Match (EM) on Intra-Domain Datasets (%). *Large* denotes the base Deepseek-Coder-6.7b model, *Small* denotes the fine-tuned Deepseek-Coder-1.3b-GPTQ model, *Both* denotes both the large and small models complete the line correctly, *Either or Both* denotes either the large model, the small model, or both, complete the line correctly, and *Upper Bound* denotes the upper bound of accuracy of combining the large and small models at token-level.

Models	Django	Flask	Spring	Android
Large	45.90	45.95	48.44	54.85
Small	40.66	43.57	42.40	48.32
Both	33.77	36.19	38.54	44.59
Either or Both	52.79	53.33	52.30	58.58
Upper Bound	56.72	56.90	55.94	60.45

- We propose a novel collaborative inference framework, which simultaneously incorporates the speculative decoding algorithm and a well-designed classifier to enhance the performance of code completion. The replication package can be found in our repository[1].
- We introduce various features relevant to code completion tasks to facilitate the classifier to effectively and adaptively integrate completion results from models.
- We construct a new line-level code completion benchmark, enabling the evaluation of intra-project and intra-domain code completion across multiple programming languages and domains.
- Experiments on both intra-project and intra-domain line-level code completion demonstrate that our approach enhances the accuracy of code completion by effectively combining the completed code from large and small models. In the meantime, our approach significantly reduces the inference time and space overhead. Compared with the SOTA method, our approach achieves better performance with much less time cost and space usage, enabling users to efficiently use LLMs for domain-specific code completion.

2 MOTIVATION

Fig. 1 presents a line-level completion case from *Campus* project within *Spring* domain. As the figure illustrates, we need to complete the code to invoke the API of *DictUtils*. To implement this, we select deepseek-coder-6.7b-base [15] as the large model and fine-tune deepseek-coder-1.3b-base-GPTQ [1] with the code corpus of the corresponding project as the small model. We utilize the models to complete the line of code starting with *DictUtils*, respectively. The completion results indicate that both the large model and FT2Ra [16], one of the state-of-the-art code completion methods, generate incorrect API invocations, i.e., *clearDictCache*. The small model, although correctly completing the *setDictCache* API, generates incorrect function parameters. However, we find that if the correct API *setDictCache* is concatenated after the original code to be completed, the large model can correctly complete the function parameters. Inspired by the motivating example, we hypothesize that the large and small models have complementary capabilities in domain-specific code completion. For example, the large model may not be able to accurately recommend domain-specific APIs, while the small model fine-tuned with domain-specific data may be prone to overfitting.

To further verify the effectiveness of combining large and fine-tuned small models to improve accuracy, we conduct preliminary experiments. In detail, we construct domain-specific line-level code completion datasets (see details in Section 4.1) and evaluate the performance of both the large model and the fine-tuned small model. The experimental results shown in Table 1 indicate that the 6.7b large model and the fine-tuned 1.3b-GPTQ small model exhibit different performances in domain-specific code completion. Nevertheless, we can observe that the

[1]https://github.com/skeetyu/Domain-Specific-Code-Completion

Just Accepted

Fig. 1. Code Completion Examples

Just Accepted

performance of the models complements each other. For instance, on the *Django* dataset, the proportion of cases where both the large and small models make correct predictions is 33.77%. However, the proportion of cases where either the large model, the small model, or both, correctly complete the code, reaches 52.79%, significantly surpassing the EM achieved using a single model for completion. In other words, if we can select the correct completion between the large and small models' predicted results for each case, we can achieve substantial improvement in completion accuracy. Moreover, by dynamically leveraging the complementary strengths of both models at each token generation step, we can potentially achieve a higher upper bound of the EM metric. For instance, we may tend to select the predicted results of the large model for general tokens and choose the predicted results of the small model for domain-specific API tokens. As the experiment results present, if we can achieve optimal token-level selection during code completion, it will yield an EM score of 56.72%.

In summary, the example and piriliminary experiments demonstrate the complementarity between the large and small models in code completion, encouraging us to explore how to combine their inference capabilities during token-by-token completion efficiently. Therefore, we propose a collaborative inference framework that concurrently leverages two models for code completion, incorporating a specialized classifier to combine the predicted tokens of the models.

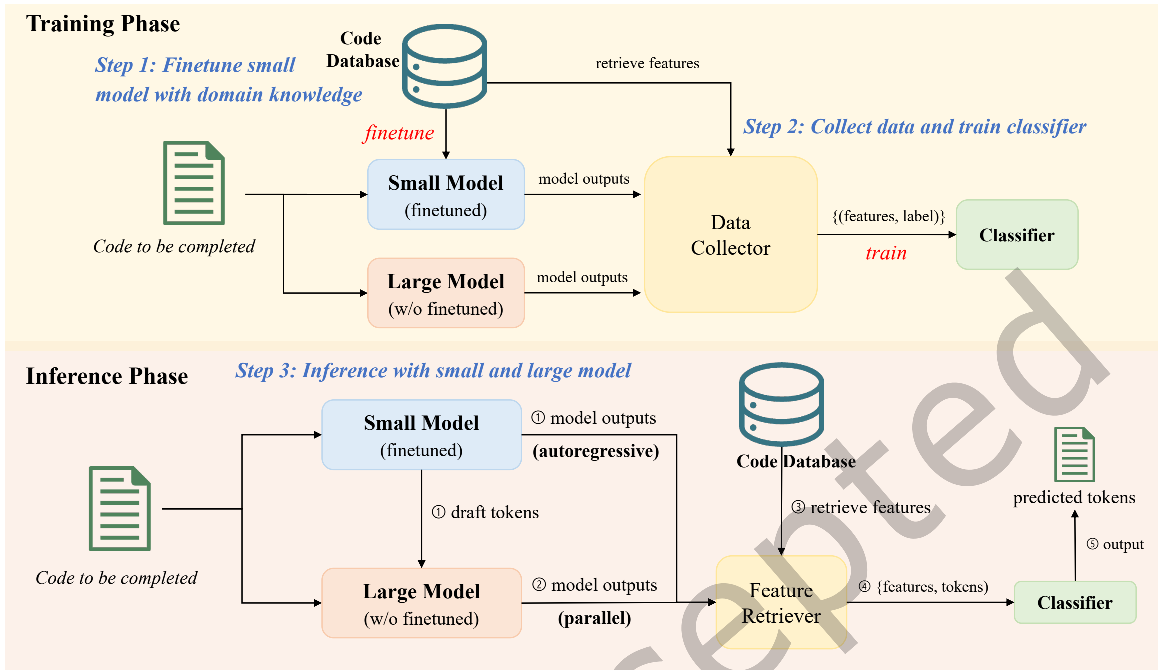

Fig. 2. The Primary Workflow of Our Approach

Just Accepted

3 APPROACH

Our work focuses on line-level code completion, which aims to predict the next line of code or complete the current line of code given a known code context $c_t = (x_1, x_2, \ldots, x_{t-1})$. During generation, the code model generates x_t based on c_t, and then concatenates x_t to form c_{t+1}, which in turn generates x_{t+1}. This process continues iteratively until the completion of a line of code.

For domain-specific code completion, we fine-tune a small model for learning domain knowledge and combine the small model's predictions with the large model. The primary workflow of our approach is illustrated in Fig. 2. We propose a collaborative inference framework based on the speculative decoding framework [24, 25] with a large model and a small model, and train a domain-specific classifier to decide which model's predicted token to select.

Our approach consists of training and inference phases.

- In the training phase, we fine-tune a small model with domain knowledge (Section 3.1) and train the domain-specific classifier (Section 3.2).
- During the inference phase (Section 3.3), we predict tokens of code using both the fine-tuned small model and large model following the procedure of speculative decoding. Then, we utilize the outputs of both the large and small models to invoke the classifier, which helps retrieve the final code completion.

3.1 Step 1. Fine-tuning small model

We fine-tune the small model using code within each domain to enhance its performance in completing line-level code specific to the particular domain. The overhead associated with fine-tuning a smaller model with fewer parameters is significantly lower compared to fine-tuning a large-scale code model [28].

Specifically, we utilize code from the training set (c.f. Section 4.1) to construct a fine-tuning dataset. We build the fine-tuning dataset $\mathcal{D} = \{(\text{input}_1, \text{output}_1), \ldots, (\text{input}_N, \text{Output}_N)\}$ (N denotes the total number of training samples) by randomly splitting each code into input $= c_t = \{x_1, x_2, \ldots, x_t\}$ and output $= \{y_{t+1}, y_{t+2}, \ldots, y_{t+L}\}$. The input is the code context, the output is the code to be completed based on the input, and L represents the total number of tokens to be completed. We fine-tune the model with the next-token prediction task, where the next token y_{t+l} is predicted based on the existing token sequence, i.e., $x_1, x_2, \ldots, x_t, y_{t+1}, \ldots, y_{t+l-1}$. The fine-tuning objective is to minimize the cross-entropy loss:

$$L(\theta) = -\frac{1}{N} \sum_{n=1}^{N} \sum_{l=1}^{L} \log P_\theta(y_{t+l}|c_t, y_{<t+l})$$

where P_θ denotes the probability distribution parameterized by the neural network with trainable parameters θ.

In addition, to further reduce the cost of fine-tuning, we choose to fine-tune the small model using the LoRA (Low-Rank Adaptation) method [19], which is one of the state-of-the-art PEFT (Parameter-Efficient Fine-Tuning) methods [35, 59]. Instead of full fine-tuning on all pre-trained model parameters, whose cost is very high and expands as the size of pre-trained language models grows, PEFT methods enable efficient adaptation of large pre-trained models to various downstream applications by only fine-tuning a small number of model parameters. Previous studies have found that, compared with full fine-tuning, the PEFT methods do not affect the code model's performance much and may achieve comparable or higher performance than standard full fine-tuning in several code-related tasks [28].

3.2 Step 2. Training Domain-Specific Classifier

We concurrently employ both the large model and the fine-tuned small model for inference to complete code snippets. In cases where the completion results from the large and small models differ, we leverage the outputs of both models to extract relevant information and features, which are then fed into a well-designed classifier to determine which model's predicted tokens to trust.

In this part, we first discuss the construction of training data for the classifier (Section 3.2.1. Then, we will delve into the features extracted as inputs for the classifier (Section 3.2.2).

3.2.1 Data Collection. Before discussing our construction of the classifier's training data, we define two distinct events associated with predicting the next token using the large model and fine-tuned small model:

Event E: The fine-tuned small model predicts the next token correctly, and the large model predicts the next token incorrectly.

- Set notation:

$$E = \{\arg\max(p_s(y_s|c_t)) = x_t \text{ and } \arg\max(p_l(y_l|c_t)) \neq x_t\}$$

- Description: The predicted token y_s with the highest probability from the fine-tuned small model matches the ground truth x_t, and the predicted token y_l with the highest probability from the large model does not match the ground truth x_t.

Event E': The fine-tuned small model predicts the next token incorrectly, and the large model predicts the next token correctly.

Just Accepted

Table 2. Features Collected in Our Approach

Dim	Feature	Definition
Probability	PT	The output probabilities of tokens generated by both the large and small model at the position
Type	TW	The type of word generated by the model at the position
Length	LEN	The length of both the currently completed line and the line preceding it
Frequency	FW	The frequency of occurrence of the currently generated word within the code context
	FL	The frequency of occurrence of both the currently completed line and the snippet consisting of the preceding line and the currently completed line, within the code context
Similarity	SIM	The similarity of both the currently completed line and the snippet consisting of the preceding line and the currently completed line, to the code context
Retrieval	RF	The frequency of occurrence of both the currently completed line and the snippet consisting of the preceding line and the currently completed line within the code corpus
	RS	The similarity of both the currently completed line and the snippet consisting of the preceding line and the currently completed line, to the code corpus

- Set notation:

$$E' = \{\arg\max(p_s(y_s|c_t)) \neq x_t \ \text{ and } \ \arg\max(p_l(y_l|c_t)) = x_t\}$$

- Description: The predicted token y_s with the highest probability from the fine-tuned small model does not match the ground truth x_t, and the predicted token y_l with the highest probability from the large model matches the ground truth x_t.

The inputs for our procedure of data collection are various contexts like c_t. We respectively utilize the large and fine-tuned small models to accomplish code completion tasks, predicting the next token x_t given the code context c_t. For tokens belonging to event E, we assume that the fine-tuned small model has acquired relevant domain-specific knowledge, enabling it to complete accurately. For tokens belonging to event E', considering that the fine-tuned small model cannot correctly predict these tokens, while the large model can, we should strictly follow the predictions from the large model when predicting these tokens. We ignore the examples where the large and small models predict both correctly or incorrectly because these cases will not affect the correctness of the final predictions. In cases where both models output correct results, there is no need to invoke the classifier for decision-making. If both models predict incorrectly, generating subsequent tokens based on these errors, regardless of which model's prediction we accept, would be meaningless.

For each completion case that satisfies event E or event E', we gather relevant information to form the output. Specifically, we label one for those belonging to event E and zero for those belonging to event E'. We store *embedding*, i.e., the input of the last layer in model architecture following [45], [16], which is considered to contain useful information about the current completion. Furthermore, we extract various *features* based on the outputs from both models (see details in Section 3.2.2) to help the classifier decide which token to choose. So far, we construct the training data $\{(\text{embeddings}, \text{features}, \text{label})\}$ for our domain-specific classifier.

3.2.2 Studied Features. We extract features only when either the large model or the fine-tuned small model predicts the next token correctly, utilizing the outputs from each model. These features reflect the model's confidence in the current prediction to some extent, helping us decide which model's prediction to trust.

Just Accepted

Overall, we analyze features related to code completion from six dimensions, which are *Probability* and *Type* of the predicted token output by the code model, *Length* of the model's generation or the code context, the *Frequency* of occurrence of the currently completed results within the code context, and *Similarity* between the currently completed results and the code context. In addition, we define a *Retrieval* dimension feature that involves retrieving corresponding frequency and similarity values from the code corpus. Table 2 summarizes all the features we investigate.

We investigate *Probability* feature, as the probability value provides insights into the model's confidence in its current predictions [55] and helps the classifier consider which model's predicted token is more reliable. We postulate that the higher the probability of the model output, the more likely that the predicted token will be correct. Note that we collect not only the probability of the token generated by the corresponding model itself, but also the probability of the token predicted by another model.

Regarding the *Type* feature, we classify the types of tokens generated by the models, considering that the models exhibit different inference capabilities on tokens associated with different types [45]. For instance, previous studies have demonstrated that code in specific domains follows unique naming conventions [5], which presents a significant challenge in domain-specific code completion [22, 40]. In this circumstance, we assume that the fine-tuned small model has a higher probability of successfully predicting such tokens, as the model has learned sufficient knowledge. In detail, we classify tokens into six types, namely the empty type, numeric type, keyword type, punctuation type, identifier type, and a special type. Specific tokens such as <EOS> (end-of-sentence) and <EOL> (end-of-line), and tokens that can not be directly regarded as other types are classified as the special type. Specifically, for each subtoken, we determine its type based on the type of the word formed by it. We determine the type of each word as follows: we classify the words that are empty or start with numeric or punctuation subtokens as empty, numeric, or punctuation, respectively. Each word starting with a letter is classified as either a keyword or an identifier.

The last three dimensions of features, *Frequency*, *Similarity*, and *Retrieval*, measure the correlation between the current completion and code within the domain. The intuition for investigating these features is that the code to be completed often has some presence in the preceding code context, such as commonly used variables. The higher the frequency and similarity of the completions within the preceding context, the more likely it is to be the correct completion. Besides, a lot of research [29, 31, 51] has demonstrated the effectiveness of retrieving similar code snippets to assist code completion models, allowing the code model to learn and mimic these similar snippets for effective completion. Therefore, we measure the correlation between current completed results and code within the domain, assuming that a higher frequency of occurrence or a higher similarity with the code in the domain's code corpus indicates a higher likelihood of accuracy for the current code completion. In detail, for the *Frequency* feature, we calculate the frequencies of occurrence for the currently generated subtokens, words, and statements composed of all the current predicted words. Particularly, since a word may be composed of multiple subtokens, when calculating the occurrence frequencies, we not only conduct a complete character match for the entire word but also perform prefix matching based on the subtokens. Words that begin with the target subtoken will be incorporated into the frequency calculation. For instance, when calculating the occurrence frequency of the subtoken "Dict", words such as "Dict" and "DictType" will all be included in the calculation.

For the *Similarity* feature, we utilize the BM25 similarity algorithm [38] to measure the relevance between the current completion and the code context. In addition, we combine the last line of code context and the current completion as one code snippet, calculating their frequency of occurrence and similarity with the code context.

According to the *Retrieval* feature, we extract features by retrieving from code in the training set, using the current completion itself, as well as the combined code snippet with the preceding context. Specifically, we calculate the occurrence frequencies and Jaccard similarity [21] scores with similar code snippets in the code corpus to measure the similarity between the current completion content and the code corpus. It is worth noting that different similarity predicates have varying strengths [57]. We use BM25 to more precisely measure

Just Accepted

the similarity between the current completion results and the code context while utilizing Jaccard for a more comprehensive retrieval within the corpus [4].

We also collect the *Length* of the content already generated by the model as an additional feature. The *Length* feature reflects other features to some extent. For example, when the length of currently generated content equals one, i.e., each model only outputs one token, the occurrence frequency feature of the currently generated statements is the same as that of the generated tokens, resulting in redundancy.

3.2.3 Classifier Training. We train a multi-layer MLP neural network as the classifier using data collected following the procedure described in Section 3.2.1. Specifically, for features like *frequency*, *similarity*, and *retrieval*, we calculate the difference between the corresponding features from the large and small models and incorporate them as inputs into the classifier. And for the *embeddings* and features of *type* and *length*, we directly feed them into the classifier model because subtracting values of features like *type* is meaningless.

It is worth noting that our classifier will only predict which model's predicted token to trust based on the embeddings and features, and will not output the actual code. Therefore, even training a shared classifier for all domains will not cause cross-domain code recommendation problems and is expected to improve generalization.

3.3 Step 3. Inference Phase

During the inference phase, we leverage the speculative decoding method to complete the predictions. Algorithm 1 generalizes the procedure to generate tokens from position t to $t + l - 1$ at once.

In detail, we first utilize the fine-tuned small model to predict tokens autoregressively, given the context $c_t = (x_1, x_2, \ldots, x_{t-1})$. The small model outputs $\{(\text{output}_t,\ y_t), \ldots, (\text{output}_{t+l-1},\ y_{t+l-1})\}$ where output_t represents the *embedding* (mentioned in Section 3.2.2) and logits output by the model, and y_t denotes the token predicted by the model.

Then we employ the large model to output in parallel based on tokens predicted by the small model, i.e., $\{(\text{output}'_t,\ y'_t), \ldots, (\text{output}'_{t+l-1},\ y'_{t+l-1})\}$. It is worth noting that here, the large model generates outputs simultaneously at l positions, significantly enhancing the model's inference speed compared to sequential generation.

After generation, we compare the predicted results of the large and small models and combine them with the help of the classifier to choose which model's predicted token to accept at each position. In detail, for each position i, the small model outputs $(\text{output}_i,\ y_i)$ and the large model outputs $(\text{output}'_i,\ y'_i)$. When the small and large models predict the same token, i.e., $y_i = y'_i$, we naturally accept the token. Inversely, when the small and large models output different tokens, we invoke the classifier to decide which model's predicted token to trust.

Specifically, we first invoke a feature retriever, which takes $\text{output}_i,\ \text{output}'_i$, the preceding code context, and the code corpus as inputs to extract all the features mentioned in Section 3.2.2. We do not invoke the feature retriever for every token, but only when the large and small models disagree on token predictions. Besides, we do not execute the feature retrieval process in parallel, because some features depend on the verification results of the prior tokens. After feature retrieval, the domain-specific classifier will make decisions based on the *embeddings* and all the extracted features and output the corresponding label.

- If the label equals 1, we trust the small model's prediction y_i. Then we move on to the next position $i + 1$.
- Conversely, if the label equals 0, we reject the small model's predicted token y_i and accept y'_i. Then, we discard all the remaining tokens to end the iteration, for all the remaining tokens $[y'_k \mid k > i]$ are predicted based on the rejected token y_i.

Finally, we end up with all accepted tokens as the output of this iteration and repeat this process until the model completes a line of code or the output length reaches the maximum limit.

Fig. 3 further depicts a running example of our code completion procedure. In this example, the small model first generates tokens of <old>, <D>, <ict>, <Type>, and <=> autoregressively. Then the large model generates tokens of <sys>, <D>, <ict>, <=>, and <=> in parallel, based on the input code context and the small model's

Just Accepted

Algorithm 1 Code Completion Procedure

Inputs: The small model M_s, the large model M_l, code context $c_t = (x_1, x_2, \ldots, x_{t-1})$, maximum draft length l, the classifier M_c
▷ 1. Sample l tokens(y_i) using M_s autoregressively
for $i = t$ **to** $t + l - 1$ **do**
  $\text{output}_i,\ y_i \leftarrow M_s(x|c_t)$
  $c_t \leftarrow c_t + y_i$
end for
▷ 2. Sample l tokens using M_l in parallel
$[\text{output}'_i,\ y'_i | t \leq i < t + l] \leftarrow M_l(x|x_1, \ldots, x_{t-1}, y_t, \ldots, y_{t+l-1})$
▷ 3. Verify tokens to get final outputs with classifier
for $i = t$ **to** $t + l - 1$ **do**
  if $y_i = y'_i$ **then**
    ▷ Accept the same token
    continue
  else
    ▷ Invoke classifier to decide which model's predicted token to select
    ▷ Retrieve features when the large and small models predict different tokens
    $f_i, f'_i \leftarrow$ feature retriever $(\text{output}_i, \text{output}'_i)$
    label $\leftarrow M_c(f_i, f'_i)$
    if label = 1 **then**
      ▷ Accept y_i predicted by the small model, then verify tokens of the next position
      continue
    else
      ▷ Accept y'_i predicted by the large model, then end current iteration
      return $[y_t, \ldots, y_{i-1}] + y'_i$
    end if
  end if
end for
▷ If all tokens generated by the small model are accepted
return $[y_t, \ldots, y_{t+l-1}]$

Just Accepted

predictions. After generation, we invoke the classifier to verify the tokens on which the large and small models disagree.

As Fig. 3 presents, we extract features and invoke the classifier for the first token, as the small model generates <old> while the large model generates <sys>. Here, we accept <old> generated by the small model because the classifier outputs a label of 1. Then we proceed to conduct verification on the subsequent tokens until we choose the predicted token from the large model for the fourth token. At this point, we opt to accept <=> generated by the large model and terminate this iteration. The final output obtained from this round of generation is "oldDict =".

4 EXPERIMENTAL SETUP

To assess the efficacy of our approach, we formulate the following research questions:

- RQ1: How does our approach perform in intra-project code completion?

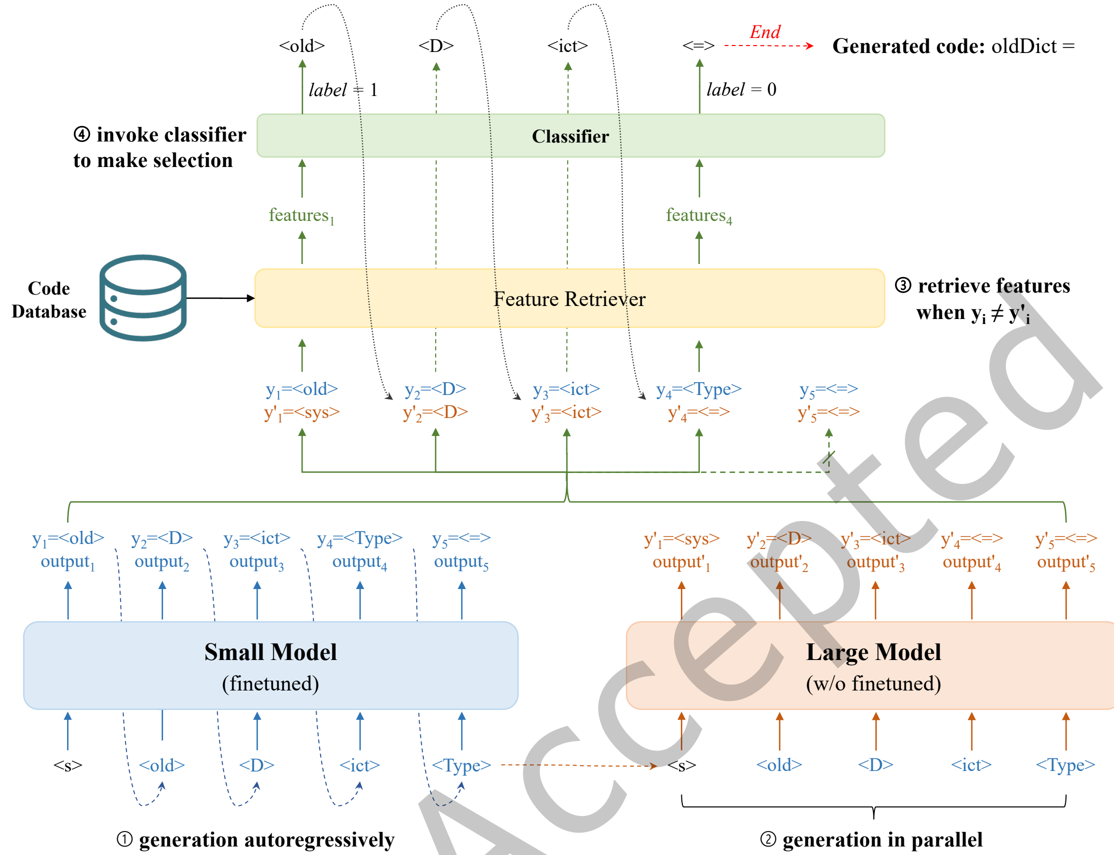

Fig. 3. Running Example of Our Code Completion Procedure Described in Algorithm 1

- RQ2: How does our approach perform in intra-domain code completion?
- RQ3: How is the time and space efficiency of our approach compared to the baselines?
- RQ4: How much does each feature contribute to the accuracy of the classifier?

4.1 Benchmark

To evaluate the performance of our approach in domain-specific code completion, we construct four intra-domain and twelve intra-project code completion datasets by collecting open source projects on GitHub using the GitHub Rest API [2]. We do not directly use the datasets constructed in previous works like [16], because the large language models may have already encountered the data in these datasets during pre-training. The datasets' statistics are displayed in Table 3, 4.

Projects Collection. To collect open source projects belonging to different domains, we select four domains, two in Java and two in Python. For the Python language, we select *Django* and *Flask*, which are the most popular web development frameworks. For the Java language, we choose *Spring*, as well as *Android*, which are both widely recognized in the Java community. *Spring* is renowned as a popular web development framework, while *Android*

Just Accepted

Table 3. Statistics of Our Intra-Domain Code Completion Dataset

Domain	#Projects	Train	Valid	Test
Django	13	487	55	55
Flask	16	560	59	59
Spring	10	1863	215	215
Android	15	743	86	86

[1] Numbers in the Train, Valid, and Test columns refer to the number of code files in the corresponding set.

Table 4. Statistics of Our Intra-Project Code Completion Dataset

Project	Train	Valid	Test	Project	Train	Valid	Test
Django-ninja-crud	50	6	6	CoinExchange	324	36	36
Django-vue3-admin	48	6	6	Ddd-boot	296	37	37
Falco	47	5	5	Eladmin-mp	196	21	21
Applio	57	6	6	WaEnhancer	81	10	10
Claude2-PyAPI	46	5	5	BiliSendCommAntifraud	72	8	8
Feedi	42	5	5	Accord	63	7	7

[1] Numbers in the Train, Valid, and Test columns refer to the number of code files in the corresponding set.

is recognized as a leading mobile development framework, each boasting a rich ecosystem of notable open-source projects.

For each domain, we search and filter projects belonging to that domain. Specifically, for each domain, we search for all corresponding projects where the domain keyword appears in either the name, description, topics, or file of *readme*. Furthermore, we validate these projects from various perspectives to ensure they are reliable projects within each specific domain.

- We manually inspect the code from each project to ensure that these projects use the relevant domain-specific libraries.
- We filter out the projects created before February 1, 2023, ensuring the code selected has not been previously learned by code models like DeepSeek-Coder, to mitigate potential data leakage.
- We exclude projects with fewer than 100 stars to filter out potentially low-quality projects.
- We remove projects with either too few or too many source code files, as projects with too few code files do not provide sufficient domain-specific code dependencies [30], while projects with too many code files may introduce bias and dominate the dataset [43].

We further remove the bottom 25% and top 25% of the projects based on the number of files within each domain, aiming to balance the size differences among projects and mitigate bias.

Data Process. After collecting the projects, we process the code in each project and construct the corresponding line-level completion test sets. First, we acquire non-empty Python or Java files from each project within different domains. Code files of each project are partitioned into train, validation, and test data in approximately a ratio of 80:10:10 following prior work [34]. Specifically, we calculate the Jaccard similarity between code files within the same project. If one code file in the test set has a Jaccard similarity of over 70% with another code file in the training set, we consider them excessively similar and thus repartition them.

Just Accepted

To evaluate intra-project code completion, we select the top three projects with the highest number of files within each domain, assuming that projects with a larger number of files and lines of code are more likely to encompass more domain-specific knowledge. To evaluate intra-domain code completion, we combine the training, validation, and test sets of each project in each domain to form the training, validation, and test sets specific to each domain. Finally, we construct the line-level code completion test sets for each project or domain, following instructions in prior work [3, 34].

4.2 Baselines

We compare our approach with the large and fine-tuned small models themselves, as well as the state-of-the-art domain-specific code completion approaches FT2Ra and kNM-LM.

- **FT2Ra** [16]: It develops a retrieval-based method aiming to mimic genuine fine-tuning by adopting a paradigm with a learning rate and multi-epoch retrievals.
- **kNM-LM** [45]: It utilizes the in-domain code to build the retrieval-based database decoupled from the code model and then combines it with the code model through Bayesian inference to complete the code.

Furthermore, since our approach implements a speculative decoding pipeline and employs a trained classifier to verify the sampling results of both large and small models, we also compare our work with other speculative decoding approaches that utilize different verification strategies.

- **Raw-SD** [25]: The state-of-the-art speculative decoding studies employ probabilistic acceptance for small model's outputs. The method will accept predictions from the small model with a certain probability calculated based on the discribution calculated by the large and small models. Once outputs generated by the small model are rejected, the method will calculate a residual distribution based on the probability distribution from the large and small models for resampling.
- **Bild** [24]: The Bild framework introduces fallback and rollback policies to leverage the large and small models for collaborative inference. Its fallback mechanism determines when to employ the large model for inference, and the rollback verification strategy rejects small model outputs when their distance from the large model exceeds a predefined threshold.

4.3 Metrics

We evaluate our approach and the baselines on two metrics, EM and ES, following [16], which are widely used to evaluate the performance of line-level code completion [9, 44]. Exact-Match (EM) counts the fraction of exactly correct predictions and can most intuitively reflect the accuracy of different approaches. Edit Similarity (ES) is a kind of edit distance and measures the similarity between the model's prediction and the correct code. In many cases, even if the code model only predicts an approximate line of code, developers are willing to adopt the prediction with modest edits.

4.4 Implementation Details

We choose Deepseek Coder [15], with a wide range of code models of different sizes, all capable of supporting code completion tasks, as our base model. Specifically, we use the Deepseek-Coder-6.7b-base model as our large model and fine-tune Deepseek-Coder-1.3b-base-GPTQ (Generative Pre-trained Transformer Quantization) version [1] as our small model.

During the training phase, we fine-tune the small model deepseek-coder-1.3b-base-GPTQ on the train data for each domain (for RQ1) or project (for RQ2), enabling the model to learn sufficient domain knowledge. In detail, we use the LoRA method, one of the state-of-the-art PEFT methods, to fine-tune the small model. When training the classifier, we use a two-layer MLP (Multilayer Perceptron) neural network as the classifier model.

Just Accepted

Table 5. Results of Intra-Project Line-Level Code Completion (%). *large* denotes the base Deepseek-Coder-6.7b model and *small* denotes the fine-tuned Deepseek-Coder-1.3b-GPTQ model.

Datasets	Large		Small		Raw-SD		Bild		kNM-LM		FT2Ra		Ours	
	EM	ES	EM	ES	EM	ES	EM	ES	EM	ES	EM	ES	EM	ES
Django-ninja-crud	54.37	82.26	53.40	81.74	48.54	80.49	56.31	83.04	55.34	82.41	57.28	84.60	**58.25**	**84.74**
Django-vue3-admin	36.26	75.82	38.46	75.19	34.07	74.22	37.36	**78.54**	36.26	76.21	38.46	76.78	**40.66**	75.80
Falco	25.71	71.49	28.57	72.83	17.14	68.26	25.71	71.94	25.71	72.00	28.57	71.37	**31.43**	**75.11**
Applio	36.67	75.31	38.89	74.58	28.89	70.33	41.11	76.27	38.89	76.52	40.00	76.26	**44.44**	**77.29**
Claude2-PyAPI	38.97	**77.29**	41.91	75.91	41.18	74.01	41.91	75.72	38.97	76.57	36.03	74.09	**42.65**	77.17
Feedi	26.92	71.87	21.79	67.03	26.92	67.91	**29.49**	71.81	26.92	71.38	28.21	71.05	**29.49**	**72.86**
CoinExchange	48.03	83.67	47.64	81.18	44.09	79.47	50.00	83.60	48.03	83.82	47.64	**84.21**	**50.79**	84.15
Ddd-boot	43.68	81.62	36.02	74.39	36.02	76.97	44.06	80.80	43.68	81.61	44.44	81.77	**46.74**	**82.95**
Eladmin-mp	55.61	87.08	44.86	82.12	50.93	83.73	57.48	87.02	56.07	87.17	57.48	87.96	**58.88**	**88.90**
WaEnhancer	45.83	82.02	37.50	75.81	33.33	76.00	41.67	78.11	45.83	82.06	46.67	80.08	**48.33**	**82.56**
BiliSendCommAntifraud	47.66	83.19	39.25	79.36	42.99	79.59	47.66	81.30	48.60	83.99	45.79	82.66	**48.60**	**84.26**
Accord	54.81	82.98	38.46	73.77	48.08	77.33	51.92	80.36	55.77	82.96	49.04	82.50	**55.77**	**83.13**
Avg	42.88	79.55	38.90	76.16	37.68	75.69	43.72	79.04	43.34	79.73	43.30	79.44	**46.34**	**80.74**

To enable comparison with domain-specific code completion baselines, we utilize the existing code provided by FT2Ra [16], with necessary adaptations to accommodate the Deepseek-Coder models. It is worth noting that the implementation of kNM-LM leverages a GPU to accelerate retrieval, while FT2Ra does not. To conduct a fair comparison with FT2Ra, we also use FT2Ra with GPU acceleration. As for the speculative decoding baselines, we reimplement the verification strategies based on the algorithms described in these works. Specifically, for Bild [24], to ensure higher generation quality, we set the fallback threshold to 0.6 and the rollback threshold to 2 according to the empirical findings in its paper.

5 EVALUATION RESULTS

5.1 RQ1 Intra-project code completion

In real-world software development, the intra-project code completion scenario is prevalent, assisting developers in efficiently completing code within a specific project. In this research question, our goal is to evaluate the performance of our approach in intra-project code completion and demonstrate the positive impact achieved by fine-tuning smaller models to learn domain-specific knowledge and combining the sampling results of the large and small models through our classifier.

Table 5 presents the performance of our method along with various baselines in intra-project code completion. From the results, we can observe that our approach outperforms all the selected baselines in terms of EM, demonstrating the effectiveness of our strategy in combining the sampling results of large and small models. On average, our approach achieves a 9.13% improvement in the exact match (EM) compared to the large model and a 7.42% improvement compared to FT2Ra.

From the results, we can also observe that the performance of different methods varies across different datasets. One possible reason may be attributed to the code structure within each project. With fewer code files in a single project, the dependencies and connections between different code files are weaker, which in turn may negatively impact the performance of retrieval-based methods. In contrast, our approach maintains a certain level of accuracy by balancing the sampling results of large and small models through the utilization of our classifier. Besides, our approach outperforms existing speculative decoding baselines that prioritize speed over accuracy. Unlike their verification strategies, which typically favor the large model's outputs, our method achieves higher precision through our classifier-driven mechanism that balances different models' outputs.

Just Accepted

To better illustrate that our approach can help the large and small models complement each other, we draw the Venn diagram of the number of test samples that each approach can correctly handle in Fig. 4. For convenience, we refer to the test samples that only the large model or small model can correctly complete (i.e., exact match) as potential samples. It can be observed that our approach can correctly handle 76.3% (212/278) of the samples and achieves better performance than the large and small models. In addition, our approach can correctly complete 16 samples that both the large and small models fail to handle, bringing additional accuracy improvement. In these cases, the large and small models both incorrectly predict some tokens, but there is no overlap between the two token sets. By accurately identifying the appropriate tokens predicted by the two models, our approach effectively addresses these cases.

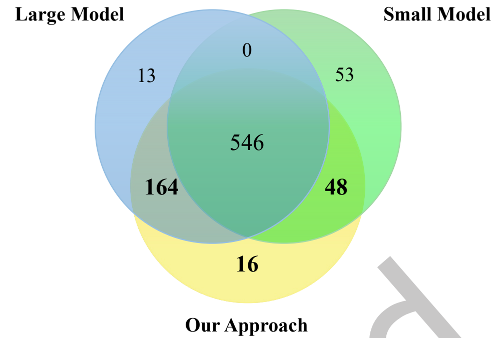

Fig. 4. Venn Diagram of the Exact Match on Overall Intra-project Code Completion

Furthermore, we calculate the average number of tokens generated by the large and small models to show that the large and small models are complementary. In detail, we count the instances where token predictions differ between large and small models, which therefore require classifier intervention. Specifically, we focus only on the cases where either large or small models make correct predictions. Ultimately, we collect a total of 780 instances across all intra-project code completion datasets. In these cases, the large model correctly predicts 535 tokens, while the small model accurately predicts 245 tokens, further demonstrating the complementarity between the large and small models.

Case Study. In Fig. 5, we present an illustrative example showing the advantages of our approach that combines predictions of the large and small models. We can observe that only our approach completes the correct line, effectively combining predictions from the large and small models with the assistance of the classifier. Additionally, Fig. 6 depicts a scenario in which the large model and our approach complete the line. Despite this, both the fine-tuned model and the FT2Ra method struggle with accurately completing the line due to training data bias associated with the *product_create* function. In contrast, our method utilizes predictions from the large model, supported by the classifier, to ensure the correct completion of the line.

Analysis of Edit Similarity. In addition, in our evaluation results, an increase in the exact match does not always imply an increase in edit similarity. This is because line-level code completion requires correctly predicting multiple tokens. In cases where the base model fails to make successful predictions, our approach may rely on the sampling of tokens from the small model at a certain position through the utilization of our classifier. The subsequent completion is then based on this trusted token, which may deviate from the sampling results of the base model, resulting in a decrease in edit similarity. However, this situation does not affect the exact match, because the original completion results from the base model are already incorrect.

Just Accepted

RQ1 Our approach outperforms the base model and selected baselines on datasets from various projects, demonstrating that our approach, which combines the sampling results of the fine-tuned small model and large model through a classifier, enables us to achieve superior performance.

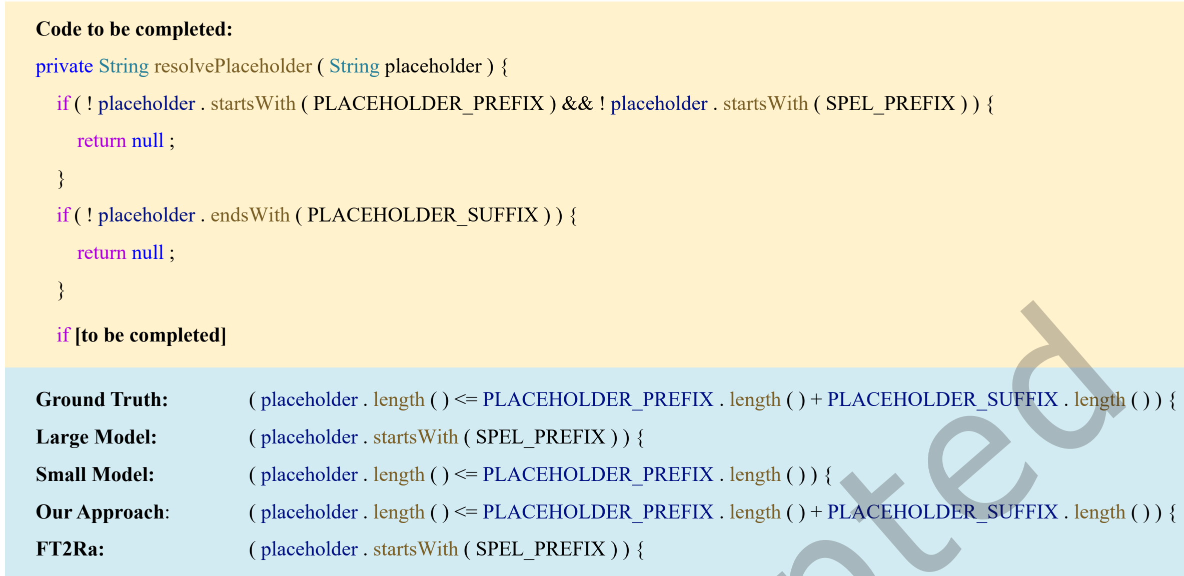

Fig. 5. Case Study for Code Completion on *Spring-Ddd-boot* Dataset

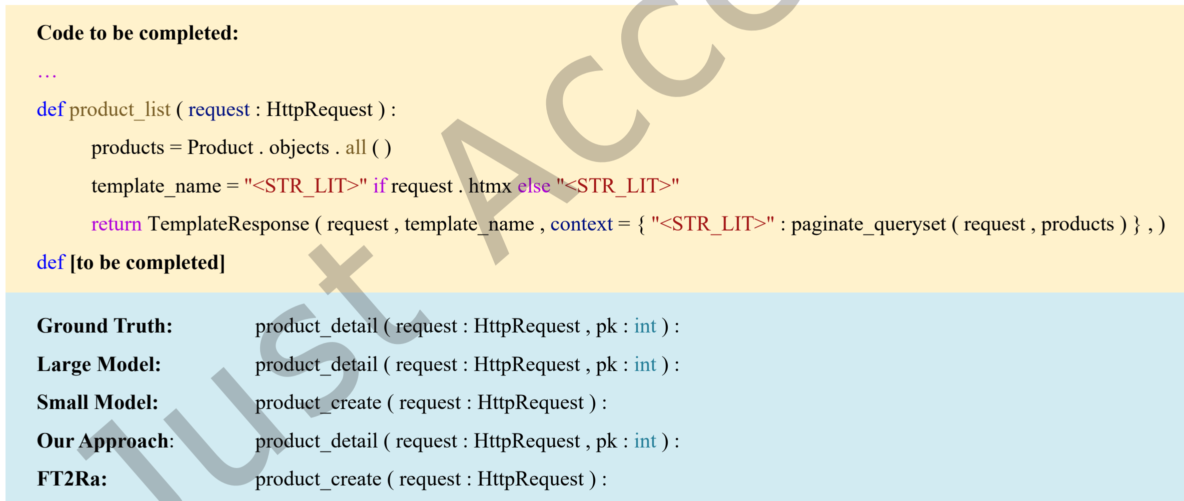

Fig. 6. Case Study for Code Completion on *Django-ninja-crud* Dataset

Just Accepted

5.2 RQ2 Intra-domain code completion

In this RQ, we evaluate the performance of our approach in intra-domain code completion. To achieve this, we utilize datasets of domains introduced in Section 4.1 and construct line-level code completion datasets using the test set from each domain, using the same method as in RQ1. Specifically, we select a fixed number of test

Table 6. Results of Intra-Domain Line-Level Code Completion (%). *large* denotes the base Deepseek-Coder-6.7b model and *small* denotes the fine-tuned Deepseek-Coder-1.3b-GPTQ model.

Datasets	Large		Small		Raw-SD		Bild		kNM-LM		FT2Ra		Ours	
	EM	ES	EM	ES	EM	ES	EM	ES	EM	ES	EM	ES	EM	ES
Django	45.90	77.32	40.33	75.10	38.03	72.03	42.30	77.89	46.23	77.65	46.23	77.54	**49.84**	**79.22**
Flask	45.95	78.17	43.33	77.58	38.10	74.27	45.71	77.83	46.19	78.28	47.14	78.61	**49.29**	**80.00**
Spring	48.44	83.80	42.19	81.25	43.44	80.65	47.81	83.46	49.17	84.12	50.00	**84.98**	**51.04**	84.69
Android	54.85	85.43	48.88	82.18	48.13	82.06	52.80	84.44	55.04	85.50	53.17	84.79	**55.41**	**85.59**
Avg	48.79	81.18	43.68	79.03	41.93	77.25	47.16	80.91	49.16	81.39	49.14	81.48	**51.40**	**82.38**

cases for each project in each domain to balance the proportion of test cases across different projects within each domain.

The evaluation results are shown in Table 6. From the table, we can observe that our approach demonstrates better performance than each baseline in each domain's dataset. For example, our approach achieves $(49.29, 80.00)$ on *Flask* domain, while the pre-trained large model, kNM-LM and FT2Ra achieve scores of $(45.95, 78.17)$, $(46.19, 78.28)$ and $(47.14, 78.61)$ respectively. On average, compared to FT2Ra, our approach achieved an average improvement of 4.67% of edit match on the test sets across the four domains, as well as a 5.57% improvement compared to the base large model.

It is worth noting that the domain-specific code completion baselines do not consistently outperform the base large model across all datasets. For instance, results on *Android* domain suggest that the EM score of FT2Ra exhibits a slight decrease when compared to the base large model. The possible reasons for this are consistent with our analysis in Section 5.1. Besides, this may be attributed to the substantial variations in code among different projects within the *Android* domain, which may cause interference with the retrieval-based methods. However, our approach effectively mitigates this issue by leveraging the sampling results from the large model through the utilization of the classifier, ensuring base-level accuracy while achieving a slight improvement by incorporating the sampling results from the small model.

RQ2 Our approach demonstrates remarkable performance in domain-specific line-level code completion tasks by leveraging smaller models to enhance performance while maintaining base accuracy, outperforming the base model and baselines significantly.

5.3 RQ3 Overhead Comparison

In this research question, we evaluate the overhead of our approach from both the time and space dimensions and compare it with the state-of-the-art domain-specific code completion methods FT2Ra and kNM-LM. Our evaluation does not include Raw-SD and Bild methods, because these approaches employ speculative decoding that inherently trade off accuracy for faster inference speeds and do not employ retrieval augmentation. And retrieval introduces non-negligible costs. Specifically, we conduct evaluations on the datasets from the four selected domains and record the average time and space overhead. For consistency, all experiments are conducted on a single NVIDIA A800 GPU.

Our evaluation results on the time overhead are shown in Table 7. To gain insights into each component's time costs, we evaluate our approach's inference time breakdown, including latencies of the large model, the small model, retrieval, and the classification model. As presented, our approach achieves a 1.4x faster average inference speed compared to FT2Ra. Notably, our collaborative inference framework, incorporating speculative decoding, achieves over 2× faster large-model inference compared to FT2Ra and kNM-LM, enabled by the parallel generation of multiple tokens. Consequently, despite introducing non-trivial overhead from the small model's

Just Accepted

Table 7. Time Overhead Comparison of Our Approach, FT2Ra and kNM-LM

Approach	Train (min)	Inference Breakdown / Case(s)				Infer / Case(s)
		Large Model	Small Model	Retrieval	Classifier	
Ours	41.8	1.06	0.7	0.27	0.002	2.03
FT2Ra	12.8	2.39	/	0.46	/	2.85
kNM-LM	14.0	2.43	/	0.28	/	2.71

Table 8. Space Overhead Comparison of Our Approach, FT2Ra and kNM-LM

Approach	Space (GB)				
	Large Model	Small Model	Classifier	Datastore	Overall
Ours	12.6	0.86	0.23	/	1.09
FT2Ra	12.6	/	/	13.38	13.38
kNM-LM	12.6	/	/	1.97	1.97

inference costs, our approach achieves overall superior inference speed compared to the selected baselines. Future optimization of the model's inference speed can further reduce the overall overhead.

Despite a faster inference speed, our approach requires a moderately longer training time. This demonstrates a trade-off where we accept marginally longer training overhead to achieve improved inference efficiency compared to FT2Ra and kNM-LM. However, we argue that model training is a one-time cost. Considering inference occurs significantly more frequently than training, our approach's faster inference speeds are beneficial in real-world usage.

Table 8 presents our evaluation results on the space overhead. Our comparison focuses on disk storage costs, including the storage costs of the small model and the classifier in our approach, and the storage cost of the retrieval datastore constructed by FT2Ra and kNM-LM. We assume that the large model is deployed in the cloud, given that 7B-scale or larger models require significant hardware resources for inference, and cloud deployment of large models has emerged as the standard approach for efficiency and responsiveness [54]. Beyond that, our approach requires only marginal storage for both the fine-tuned small model and the classifier model, resulting in significantly lower storage overhead than FT2Ra, which requires storing a huge data store for retrieval.

RQ3 In comparison to the state-of-the-art domain-specific code completion baselines, our approach has an advantage in inference time. Although the training overhead is higher, the infrequency of training makes this overhead acceptable. Moreover, our approach significantly outperforms FT2Ra in terms of space overhead.

5.4 RQ4 Ablation study of classifier

In this section, we conduct ablation experiments on each dimension of the classifier's features to demonstrate the effectiveness of our chosen features. Specifically, we select datasets from the four domains and construct the classifier's training and validation data using the respective training and validation sets within each domain. The construction procedure follows the methodology described in Section 3.2.1.

To conduct the ablation experiments, we systematically remove one dimension of features at a time. We then retrain the classifier based on the rest of the features using the collected training data and evaluate the accuracy of the validation data.

Just Accepted

Table 9. Performances of the Features of Our Classifier

Features	ours	-PT	-TW	-LEN	-FW	-FL	-SIM	-RF	-RS
Accuracy(%)	**78.31**	76.67	76.84	77.75	76.89	75.05	77.37	77.15	77.89
Features	-EMB	-EMB-PT	-EMB-TW	-EMB-LEN	-EMB-FW	-EMB-FL	-EMB-SIM	-EMB-RF	-EMB-RS
Accuracy(%)	74.76	65.38	70.79	74.03	71.20	70.45	72.79	71.02	70.95

The results of the ablation experiments on our classifier are shown in Table 9. We record the average accuracy of the classifier across all domains. From Table 9, we observe that each selected dimension of features makes a certain contribution to the classifier. As presented, our classifier achieves an average accuracy of 78.31% in the validation data of each domain, and removing any of these features results in a perceptible decrease in the accuracy.

Among all the features, the removal of the *EMB* feature results in the largest decrease in the classifier's average accuracy, dropping from 78.31% to 74.76%. This is because the embedding feature constitutes a substantial portion of the classifier's input and carries rich, semantically processed information derived from the underlying code model. Moreover, the synergistic integration of diverse features may yield substantial performance improvements for the classifier. For instance, the embedding feature and the retrieval feature may exhibit complementarity, as the embedding feature captures implicit semantic information, whereas the retrieval feature emphasizes explicit similarities.

To verify the hypothesis and mitigate the dominant effect of the embedding feature, we conduct additional ablation experiments by simultaneously removing both the embedding feature and the selected dimensional feature. The experimental results in the second row of Table 9 demonstrate that simultaneously removing both the embedding and specific features leads to a greater decrease in accuracy compared to removing only the embedding feature, confirming that these features contribute positively to the performance of the classifier. Specifically, the removal of both embedding and probability features results in a substantial accuracy decline, with the classifier's average accuracy dropping from 78.31% to 65.38%.

These results indicate that distinct feature dimensions exhibit varying degrees of contribution to the classifier's accuracy across different domains. Based on overall accuracy performance and cross-domain applicability, we select these feature dimensions for our final implementation.

RQ4 Our classifier achieves a notable level of average accuracy on the validation data. The ablation experiments, where each feature is systematically removed, provide further evidence that our selected features are effective and reasonable.

6 DISCUSSION

In this chapter, we primarily assess the effectiveness of our approach and suggest possible limitations of our approach to domain-specific code completion based on our findings.

Model Selection. The selection of the large and small code models remains restricted, as our approach is based on token-by-token completion, and the tokenizer's vocabulary of the selected small and large models needs to be consistent. In further research, we will aim to overcome this limitation and choose code models from different families as small and large models.

We employ Deepseek-Coder-6.7B as the large model and do not select larger code models like Deepseek-Coder-33B for experiments. Although larger models excel in general-purpose code completion, their performance in domain-specific completion remains limited due to insufficient domain-specific knowledge [8]. Thus, the small models fine-tuned on domain-specific datasets could also be complementary to large models, and our approach can

Just Accepted

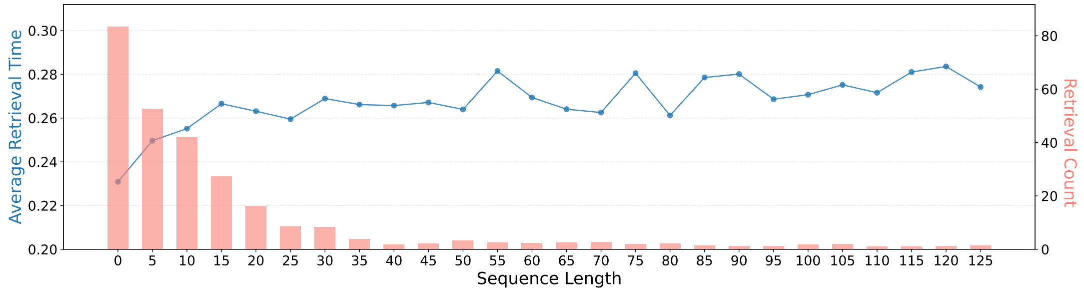

Fig. 7. Average Retrieval Time and Count under Different Sequence Lengths on *Spring* Dataset

also be applied to larger models. On the other hand, employing larger models would either significantly increase the inference latency [18] or require significantly more computation resources for deployment. Considering the effectiveness-efficiency trade-off, the model with approximately 7B parameters is usually a good choice for code completion [31]. Therefore, we argue that our model selection is reasonable. Future work may investigate the effectiveness of our approach with larger code models, as well as investigate optimal integration strategies to enhance overall code completion accuracy with an acceptable computational overhead.

Feature Selection. In our current approach, we have explored effective features from various dimensions to assist the classifier in making decisions. Our ablation experiments (Section 5.3) have also demonstrated the effectiveness of the selected features. However, during the experimental process, we discover that the contributions of different features vary across datasets for different programming languages. Therefore, we may further explore and design more useful features customized to the characteristics of different programming languages in future work.

Scalability Issue. Our approach retrieves various features based on the current completion during the token verification stage and may encounter potential scalability issues when the sequence lengths for code completion increase. Specifically, when the length of the current completion increases, the time cost of feature retrieval may also increase accordingly. To investigate such scalability issues, we conduct experiments on the *Spring* dataset, calculating the average feature retrieval time under different sequence lengths. The results are shown in Fig. 7. As presented, the average retrieval time exhibits a linear growth trend with increasing sequence length. In addition, the growth rate is relatively small, and the overall average retrieval time remains within an acceptable range between 0.20s and 0.30s. Moreover, we count the frequency of feature retrieval operations across different sequence lengths. As illustrated by the pink bars in Fig. 7, feature retrieval predominantly occurs at shorter sequence lengths, with its invocation frequency decreasing as sequence lengths increase. Therefore, we believe that as the sequence lengths increase, the total number of feature retrieval operations remains relatively stable, and the overall time overhead for feature retrieval will not increase substantially. In summary, the experimental results confirm that our approach maintains stable feature retrieval costs despite increasing sequence lengths and is applicable in long-sequence completion scenarios.

7 THREATS TO VALIDITY

Internal validity. The internal threats to validity concern the implementations of baseline methods. For the implementation of domain-specific code completion baseline methods FT2Ra and kNM-LM, we utilize the source code provided by FT2Ra [16]. However, since the original implementation is incompatible with Deepseek-Coder models, we slightly adjust the code to accommodate our chosen code model. As for the speculative decoding baseline methods Raw-SD [25] and Bild [24], we implement their verification strategies within our collaborative inference framework. We have double-checked the implementations of the baselines and inspected

Just Accepted

their experimental results to confirm the correctness of our modifications or implementations. Moreover, we have provided their implementations in our repository for being inspected by other researchers.

External validity. One of the external threats is the selection of the base pre-trained code model. We choose the Deepseek-Coder model, which has demonstrated excellent performance in code completion tasks [15], for our experiments and evaluations. Although our framework can be easily extended to other pre-trained code models, we cannot claim that our results are generalizable to arbitrary pre-trained code models. In future work, we plan to investigate the effectiveness of our approach with other pre-trained models.

Another external threat is related to the generalizability of our method, considering the selection of datasets, as there are differences between different datasets. To mitigate this threat, we select four distinct domains from two different languages, Python and Java, following recent works [16]. However, our approach is independent of specific programming languages, making it easily applicable to code completion tasks in other programming languages.

8 RELATED WORK

8.1 Code Completion

The application of machine learning technology to accomplish various tasks in software engineering is increasingly prevalent. Among these tasks, code generation tasks, including code completion tasks, stand out as a trendy area [37, 50]. Code completion tasks aim to complete the subsequent code based on the existing code context, closely simulating the daily work scenarios of software developers, and can effectively and significantly enhance the development efficiency of software engineers. Currently, a lot of LLMs designed for code-related tasks, such as CodeGeeX [56], StarCoder [26, 32], CodeLlama [39], Deepseek-Coder [15], etc., show strong code reasoning abilities and have gained great success in code completion tasks. More specifically, code completion can be divided into token-level code completion, which involves predicting the next token, and line-level code completion, which requires completing an entire line of code with contextual information [37]. Line-level code completion puts forward higher requirements for model reasoning ability, necessitating a higher accuracy for token-level prediction. Many researchers have put forward different methods on line-level code completion [14, 20, 33, 36]. Compared to these works, our work focuses more on domain-specific code completion than general-purpose code completion, focusing on domain-specific tokens that general code models can not predict correctly.

8.2 Domain-Specific Code Completion

Though LLMs perform excellent general code generation tasks, their performance degrades in domain-specific code completion tasks due to a lack of domain-specific knowledge during the completion stage [8, 12]. The conventional approach to adapting code models to specific domains is fine-tuning models. Shen et al. [40] construct a specific dataset where variable names are transformed into semantic descriptions for training code models, incorporating variable-related domain knowledge into code models. Ding et al. [11] and Shrivastava et al. [42] fine-tune code models with both code context and relevant code snippets retrieved, which contain the domain knowledge. These methods require meticulous task design to integrate domain-specific knowledge into the model effectively.

In addition to traditional fine-tuning methods, retrieval-augmented generation techniques are widely applied to domain-specific code completion tasks in various research studies. As code models have demonstrated excellent performance in code comprehension, many methods [8, 29, 33, 51] leverage the proficiency of code models to incorporate domain-specific knowledge retrieved for code completion as prompts, thereby achieving domain-specific code completion. Chen et al. [8] train an external API knowledge inquirer to acquire potentially utilized APIs, and Liu et al. [29] generate potential APIs through retrieval in an additional import statements generation phase. Despite this, the code models still struggle to precisely extract useful information from prompts, which

Just Accepted

hinders accurate intervention in the model's output through prompts. Compared to these works, our approach does not rely on retrieved information as prompts but rather condenses the retrieved information into features to assist our classifier in making decisions.

Furthermore, retrieval-augmented techniques have also been utilized to directly combine the retrieved content with the output of the model for domain-specific code completion tasks [16, 23, 45]. Tang et al. [45] propose a retrieval-augment kNM-LM framework based on kNN-LM [23], predicting domain-specific tokens by combining the probability distribution from both the model's output and that obtained through retrieval from an additional datastore. Guo et al. [16] adopt a retrieval-based mechanism with multi-iteration retrievals to mimic genuine fine-tuning for code completion. These methods require significant space overhead to store the datastore and time overheads for retrieval, and are sensitive to manually selected interpolated weights. Compared to these works, our approach employs a well-designed classifier to combine predictions from two distinct code models. Our approach does not require complex interpolated weight settings and enables precise intervention in the models' outputs.

8.3 Collaborative Decoding

Collaborative decoding of large and small models leverages the strengths of both model sizes to improve efficiency and effectiveness [53]. Recent studies [25, 48, 52] introduce the speculative decoding approach as a collaborative decoding technique, aimed at enhancing efficiency. This method involves sampling draft tokens from a small model and performing parallel verification of these tokens by a larger model. In the verification process, traditional strategies tend to favor the inference capability of the larger model, either completely trusting the larger model [52] or trusting it with a certain probability [7, 25, 49]. However, the traditional verification strategy does not guarantee the highest generation quality, as the outputs of the larger model are not always superior to the draft tokens generated by the smaller model [13, 47]. In response to the limitation, researchers have started exploring alternative verification strategies. For instance, Kim et al. [24] devise the fallback and rollback strategies, selectively choosing the sampling results from the smaller and larger models based on the cross-entropy of their probability distributions. Unlike these traditional speculative decoding approaches that merely rely on probability distribution alignment between large and small models for generation selection, our approach implements a more comprehensive verification strategy through a well-designed classifier to enhance the overall accuracy of code completion.

Recent researchers are also starting to complete tasks in the field of software engineering through collaborative decoding. Liu et al. [27] propose an acceleration scheme through speculative decoding with semantic adaptive tokens, enhancing the generation speed while maintaining nearly unchanged accuracy on code generation tasks. Chen et al. [6] introduce a novel model cascading strategy, reducing computational costs while increasing accuracy on code completion tasks. Different from these works, our approach focuses on domain-specific code completion tasks and uses a classifier to orchestrate the collaborative decoding process of models. We do not directly compare our approach with these works because [27] employs specifically configured adaptive tokens, which cannot be directly transferred to our chosen DeepSeek-Coder model for fair comparison, and [6] requires tests to guide the model collaboration, which does not apply to line-level code completion. Wang et al. [46] utilize a local code completion n-gram model to compensate for code models' domain adaptability with a classifier. However, their classifier only takes into account the probability information from the model outputs, whereas our classifier is designed based on various features specific to code completion tasks.

9 CONCLUSION

In this paper, we propose a collaborative inference framework incorporating the speculative decoding algorithm to effectively combine the completion results of large and small code models with a well-designed classifier, for better

Just Accepted

domain-specific code completion tasks. The classifier utilizes various features from different dimensions, involving probabilities and logits yielded by the models, occurrence frequencies, and similarities of the current completion, among others, to facilitate decision-making to combine the completion results of code models adaptively and effectively. We construct a new line-level domain-specific code completion benchmark to conduct evaluations. The experimental results demonstrate that our approach outperforms existing state-of-the-art methods in both intra-project and intra-domain line-level code completion by effectively combining the completed code from large and small models. Furthermore, compared with the state-of-the-art domain-specific code completion method FT2Ra, our approach achieves better performance, with a faster inference speed and significantly less space overhead, making our approach much more accessible and efficient. The ablation study also reveals that all extracted features for our classifier are reasonable and effective.

Just Accepted

ACKNOWLEDGMENTS

This research/project was supported by the National Natural Science Foundation of China (No. 62202420), Zhejiang Provincial Natural Science Foundation of China (No. LZ25F020003), and the National Natural Science Foundation of China (No. 62372398).

REFERENCES

[1] [n. d.]. Deepseek-coder-1.3b-base-GPTQ. https://huggingface.co/TheBloke/deepseek-coder-1.3b-base-GPTQ/tree/main

[2] [n. d.]. GitHub REST API. https://docs.github.com/en/rest?apiVersion=2022-11-28

[3] Miltiadis Allamanis and Charles Sutton. 2013. Mining source code repositories at massive scale using language modeling. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. 207–216. https://doi.org/10.1109/MSR.2013.6624029

[4] Fema Rose Bronda-Ecraela and Remia Doctora. 2024. Integrating Okapi BM25 and Jaccard Algorithms in Thesis Search Engine. *Journal of Innovative Technology Convergence* 6, 11 (April 2024). https://doi.org/10.69478/JITC2024v6n2a11

[5] Binger Chen and Ziawasch Abedjan. 2023. Duetcs: Code Style Transfer through Generation and Retrieval. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2362–2373. https://doi.org/10.1109/ICSE48619.2023.00198

[6] Boyuan Chen, Mingzhi Zhu, Brendan Dolan-Gavitt, Muhammad Shafique, and Siddharth Garg. 2024. Model Cascading for Code: Reducing Inference Costs with Model Cascading for LLM Based Code Generation. arXiv:2405.15842 (May 2024). http://arxiv.org/abs/2405.15842 arXiv:2405.15842 [cs].

[7] Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. 2023. Accelerating Large Language Model Decoding with Speculative Sampling. arXiv:2302.01318 (Feb. 2023). http://arxiv.org/abs/2302.01318 arXiv:2302.01318 [cs].

[8] Meng Chen, Hongyu Zhang, Chengcheng Wan, Zhao Wei, Yong Xu, Juhong Wang, and Xiaodong Gu. 2024. On the Effectiveness of Large Language Models in Domain-Specific Code Generation. arXiv:2312.01639 [cs.SE]

[9] Colin B. Clement, Shuai Lu, Xiaoyu Liu, Michele Tufano, Dawn Drain, Nan Duan, Neel Sundaresan, and Alexey Svyatkovskiy. 2021. Long-Range Modeling of Source Code Files with eWASH: Extended Window Access by Syntax Hierarchy. arXiv:2109.08780 (Sept. 2021). http://arxiv.org/abs/2109.08780 arXiv:2109.08780 [cs].

[10] Le Deng, Xiaoxia Ren, Chao Ni, Ming Liang, David Lo, and Zhongxin Liu. 2025. Enhancing Project-Specific Code Completion by Inferring Internal API Information. *IEEE Transactions on Software Engineering* 51, 9 (2025), 2566–2582. https://doi.org/10.1109/TSE.2025.3592823

[11] Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2023. CoCoMIC: Code Completion By Jointly Modeling In-file and Cross-file Context. arXiv:2212.10007 (May 2023). https://doi.org/10.48550/arXiv.2212.10007 arXiv:2212.10007 [cs].

[12] Aryaz Eghbali and Michael Pradel. 2024. De-Hallucinator: Mitigating LLM Hallucinations in Code Generation Tasks via Iterative Grounding. arXiv:2401.01701 [cs.SE] https://arxiv.org/abs/2401.01701

[13] Tao Ge, Heming Xia, Xin Sun, Si-Qing Chen, and Furu Wei. 2022. Lossless Acceleration for Seq2seq Generation with Aggressive Decoding. arXiv:2205.10350 (May 2022). http://arxiv.org/abs/2205.10350 arXiv:2205.10350 [cs].

[14] Daya Guo, Alexey Svyatkovskiy, Jian Yin, Nan Duan, Marc Brockschmidt, and Miltiadis Allamanis. 2022. Learning to Complete Code with Sketches. arXiv:2106.10158 (Jan. 2022). http://arxiv.org/abs/2106.10158 arXiv:2106.10158 [cs].

[15] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence. arXiv:2401.14196 [cs.SE]

[16] Qi Guo, Xiaohong Li, Xiaofei Xie, Shangqing Liu, Ze Tang, Ruitao Feng, Junjie Wang, Jidong Ge, and Lei Bu. 2024. FT2Ra: A Fine-Tuning-Inspired Approach to Retrieval-Augmented Code Completion. (April 2024). https://doi.org/10.1145/3650212.3652130 arXiv:2404.01554 [cs].

[17] Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Ming-Wei Chang. 2020. REALM: Retrieval-Augmented Language Model Pre-Training. arXiv:2002.08909 [cs.CL] https://arxiv.org/abs/2002.08909

[18] Zhenyu He, Zexuan Zhong, Tianle Cai, Jason D. Lee, and Di He. 2023. REST: Retrieval-Based Speculative Decoding. arXiv:2311.08252 (Nov. 2023). http://arxiv.org/abs/2311.08252 arXiv:2311.08252 [cs].

[19] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. LoRA: Low-Rank Adaptation of Large Language Models. arXiv:2106.09685 (Oct. 2021). https://doi.org/10.48550/arXiv.2106.09685 arXiv:2106.09685 [cs].

[20] Maliheh Izadi, Roberta Gismondi, and Georgios Gousios. 2022. CodeFill: multi-token code completion by jointly learning from structure and naming sequences. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 401–412. https://doi.org/10.1145/3510003.3510172

[21] Paul Jaccard. 1912. The distribution of the flora in the alpine zone. 1. *New phytologist* 11, 2 (1912), 37–50.

[22] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A Survey on Large Language Models for Code Generation. arXiv:2406.00515 (June 2024). http://arxiv.org/abs/2406.00515 arXiv:2406.00515 [cs].

[23] Urvashi Khandelwal, Omer Levy, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. 2020. Generalization through Memorization: Nearest Neighbor Language Models. arXiv:1911.00172 [cs.CL] https://arxiv.org/abs/1911.00172

[24] Sehoon Kim, Karttikeya Mangalam, Suhong Moon, Jitendra Malik, Michael W. Mahoney, Amir Gholami, and Kurt Keutzer. 2023. Speculative Decoding with Big Little Decoder. arXiv:2302.07863 (Oct. 2023). http://arxiv.org/abs/2302.07863 arXiv:2302.07863 [cs].

[25] Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast Inference from Transformers via Speculative Decoding. arXiv:2211.17192 (May 2023). http://arxiv.org/abs/2211.17192 arXiv:2211.17192 [cs].

[26] 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, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour 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! https://arxiv.org/abs/2305.06161v2

[27] Chengbo Liu and Yong Zhu. 2024. SDSAT: Accelerating LLM Inference through Speculative Decoding with Semantic Adaptive Tokens. arXiv:2403.18647 (April 2024). http://arxiv.org/abs/2403.18647 arXiv:2403.18647.

[28] Jiaxing Liu, Chaofeng Sha, and Xin Peng. 2023. An Empirical Study of Parameter-Efficient Fine-Tuning Methods for Pre-Trained Code Models. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Luxembourg, Luxembourg. https://doi.org/10.1109/ASE56229.2023.00125

[29] Mingwei Liu, Tianyong Yang, Yiling Lou, Xueying Du, Ying Wang, and Xin Peng. 2023. CodeGen4Libs: A Two-Stage Approach for Library-Oriented Code Generation. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Luxembourg, Luxembourg, 434–445. https://doi.org/10.1109/ASE56229.2023.00159

[30] Tianyang Liu, Canwen Xu, and Julian McAuley. 2023. RepoBench: Benchmarking Repository-Level Code Auto-Completion Systems. arXiv:2306.03091 (Oct. 2023). http://arxiv.org/abs/2306.03091 arXiv:2306.03091 [cs].

[31] Wei Liu, Ailun Yu, Daoguang Zan, Bo Shen, Wei Zhang, Haiyan Zhao, Zhi Jin, and Qianxiang Wang. 2024. GraphCoder: Enhancing Repository-Level Code Completion via Code Context Graph-based Retrieval and Language Model. arXiv:2406.07003 (June 2024). https://doi.org/10.48550/arXiv.2406.07003 arXiv:2406.07003 [cs].

[32] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2024. StarCoder 2 and The Stack v2: The Next Generation. arXiv:2402.19173 [cs.SE]

[33] Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung won Hwang, and Alexey Svyatkovskiy. 2022. ReACC: A Retrieval-Augmented Code Completion Framework. arXiv:2203.07722 [cs.SE]

Just Accepted

[34] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin 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. arXiv:2102.04664 (March 2021). http://arxiv.org/abs/2102.04664 arXiv:2102.04664 [cs].

[35] Sourab Mangrulkar, Sylvain Gugger, Lysandre Debut, Younes Belkada, Sayak Paul, and Benjamin Bossan. 2022. PEFT: State-of-the-art Parameter-Efficient Fine-Tuning methods. https://github.com/huggingface/peft.

[36] Anh Tuan Nguyen, Aashish Yadavally, and Tien N. Nguyen. 2023. Next Syntactic-Unit Code Completion and Applications. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 180, 5 pages. https://doi.org/10.1145/3551349.3559544

[37] Changan Niu, Chuanyi Li, Vincent Ng, Dongxiao Chen, Jidong Ge, and Bin Luo. 2023. An Empirical Comparison of Pre-Trained Models of Source Code. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2136–2148. https://doi.org/10.1109/ICSE48619.2023.00180

[38] Stephen Robertson and Hugo Zaragoza. 2009. The Probabilistic Relevance Framework: BM25 and Beyond. *Found. Trends Inf. Retr.* 3, 4 (apr 2009), 333–389. https://doi.org/10.1561/1500000019

[39] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, 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. 2024. Code Llama: Open Foundation Models for Code. arXiv:2308.12950 [cs.CL]

[40] Sijie Shen, Xiang Zhu, Yihong Dong, Qizhi Guo, Yankun Zhen, and Ge Li. 2022. Incorporating Domain Knowledge through Task Augmentation for Front-End JavaScript Code Generation. arXiv:2208.10091 [cs.SE]

[41] Ensheng Shi, Yanlin Wang, Hongyu Zhang, Lun Du, Shi Han, Dongmei Zhang, and Hongbin Sun. 2023. Towards Efficient Fine-tuning of Pre-trained Code Models: An Experimental Study and Beyond. arXiv:2304.05216 (April 2023). http://arxiv.org/abs/2304.05216 arXiv:2304.05216 [cs].

[42] Disha Shrivastava, Denis Kocetkov, Harm de Vries, Dzmitry Bahdanau, and Torsten Scholak. 2023. RepoFusion: Training Code Models to Understand Your Repository. arXiv:2306.10998 (June 2023). https://doi.org/10.48550/arXiv.2306.10998 arXiv:2306.10998 [cs].

[43] Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2023. Repository-Level Prompt Generation for Large Language Models of Code. arXiv:2206.12839 (June 2023). http://arxiv.org/abs/2206.12839 arXiv:2206.12839 [cs].

[44] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. IntelliCode Compose: Code Generation Using Transformer. arXiv:2005.08025 (Oct. 2020). http://arxiv.org/abs/2005.08025 arXiv:2005.08025 [cs].

[45] Ze Tang, Jidong Ge, Shangqing Liu, Tingwei Zhu, Tongtong Xu, Liguo Huang, and Bin Luo. 2023. Domain Adaptive Code Completion via Language Models and Decoupled Domain Databases. arXiv:2308.09313 [cs.SE]

[46] Zejun Wang, Fang Liu, Yiyang Hao, and Zhi Jin. 2023. AdaComplete: improve DL-based code completion method's domain adaptability. *Automated Software Engg.* 30, 1 (March 2023), 28 pages. https://doi.org/10.1007/s10515-023-00376-y

[47] Heming Xia, Tao Ge, Peiyi Wang, Si-Qing Chen, Furu Wei, and Zhifang Sui. 2023. Speculative Decoding: Exploiting Speculative Execution for Accelerating Seq2seq Generation. arXiv:2203.16487 (Oct. 2023). http://arxiv.org/abs/2203.16487 arXiv:2203.16487 [cs].

[48] Heming Xia, Zhe Yang, Qingxiu Dong, Peiyi Wang, Yongqi Li, Tao Ge, Tianyu Liu, Wenjie Li, and Zhifang Sui. 2024. Unlocking Efficiency in Large Language Model Inference: A Comprehensive Survey of Speculative Decoding. arXiv:2401.07851 (June 2024). http://arxiv.org/abs/2401.07851 arXiv:2401.07851.

[49] Sen Yang, Shujian Huang, Xinyu Dai, and Jiajun Chen. 2024. Multi-Candidate Speculative Decoding. arXiv:2401.06706 [cs.CL] https://arxiv.org/abs/2401.06706

[50] Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang. 2022. An extensive study on pre-trained models for program understanding and generation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 39–51. https://doi.org/10.1145/3533767.3534390

[51] 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. arXiv:2303.12570 (Oct. 2023). https://doi.org/10.48550/arXiv.2303.12570 arXiv:2303.12570 [cs].

[52] Jun Zhang, Jue Wang, Huan Li, Lidan Shou, Ke Chen, Gang Chen, and Sharad Mehrotra. 2023. Draft & Verify: Lossless Large Language Model Acceleration via Self-Speculative Decoding. arXiv:2309.08168 (Sept. 2023). http://arxiv.org/abs/2309.08168 arXiv:2309.08168 [cs].

[53] Kaiyan Zhang, Jianyu Wang, Ning Ding, Biqing Qi, Ermo Hua, Xingtai Lv, and Bowen Zhou. 2024. Fast and Slow Generating: An Empirical Study on Large and Small Language Models Collaborative Decoding. arXiv:2406.12295 (June 2024). http://arxiv.org/abs/2406.12295 arXiv:2406.12295.

[54] Mingxuan Zhang, Bo Yuan, Hanzhe Li, and Kangming Xu. 2024. LLM-Cloud Complete: Leveraging cloud computing for efficient large language model-based code completion. *Journal of Artificial Intelligence General science (JAIGS) ISSN: 3006-4023* 5, 1 (2024), 295–326.

[55] Wenrui Zhang, Tiehang Fu, Ting Yuan, Ge Zhang, Dong Chen, and Jie Wang. 2024. A Lightweight Framework for Adaptive Retrieval In Code Completion With Critique Model. arXiv:2406.10263 (June 2024). http://arxiv.org/abs/2406.10263 arXiv:2406.10263 [cs].

Just Accepted

[56] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Evaluations on HumanEval-X. arXiv:2303.17568 [cs.LG]

[57] Jia Zhu, Gabriel Pui Cheong Fung, Zeyang Lei, Min Yang, and Ying Shen. 2019. An in-depth study of similarity predicate committee. *Information Processing & Management* 56, 3 (May 2019), 381–393. https://doi.org/10.1016/j.ipm.2018.11.008

[58] Tingwei Zhu, Zhongxin Liu, Tongtong Xu, Ze Tang, Tian Zhang, Minxue Pan, and Xin Xia. 2024. Exploring and Improving Code Completion for Test Code . In *2024 IEEE/ACM 32nd International Conference on Program Comprehension (ICPC)*. IEEE Computer Society, Los Alamitos, CA, USA, 137–148. https://doi.ieeecomputersociety.org/

[59] Andrei Zlotchevski, Dawn Drain, Alexey Svyatkovskiy, Colin Clement, Neel Sundaresan, and Michele Tufano. 2022. Exploring and Evaluating Personalized Models for Code Generation. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1500–1508. https://doi.org/10.1145/3540250.3558959 arXiv:2208.13928 [cs].

Just Accepted