Enhancing Repository-Level Code Completion with Reinforcement Learning in Real-World Scenarios

Anonymous ACL submission

Abstract

The rapid development of large language models (LLMs) for code, which closely aligns with the practical needs of real-world software development, has sparked growing attention in repository-level code completion. However, existing Code LLMs present challenges in focusing on suitable contexts from the repository and in deeply reasoning about cross-file dependencies. To address these challenges, we propose a novel reinforcement learning framework for repository-level code completion. To better understand both in-file and cross-file contexts, we employ identifier-driven intent recognition to capture completion intent, thereby improving the model's performance in real-world scenarios. To enhance cross-file reasoning, we propose a reward-driven completion learning to effectively reward in complex repository completion scenarios. To guide LLMs in completing code that aligns with intended functionality and repository context, we introduce a selective-exploration strategy that directs the model to focus on low-confidence, high-reward tokens, promoting the exploration of valuable, underexplored completion patterns. Experimental results show that our approach significantly improves the performance of Code LLMs. The code and the dataset is available at https://anonymous.4open.science/r/IntentCoder-EAD5.

1 Introduction

With the rapid development of large language models (LLMs) for code (Jiang et al., 2024a; Zheng et al., 2023b), these LLMs have demonstrated advanced capabilities in code understanding and completion. Many LLMs are integrated into IDEs to support developers' daily development, one of the most common usage scenarios is code completion (Zhou et al., 2022; Wang et al., 2023a). Previous studies have explored code completion task on different granularities, such as line-level code completion (Liu et al., 2024; Izadi et al., 2022), function-level code completion (Li et al., 2022; Quan et al., 2025). More recently, the repository-level code completion (Zhang et al., 2024) has become increasingly important because it better aligns with the practical needs of developers in real-world software development. However, repo-level code completion is a non-trivial task by considering the following situations:

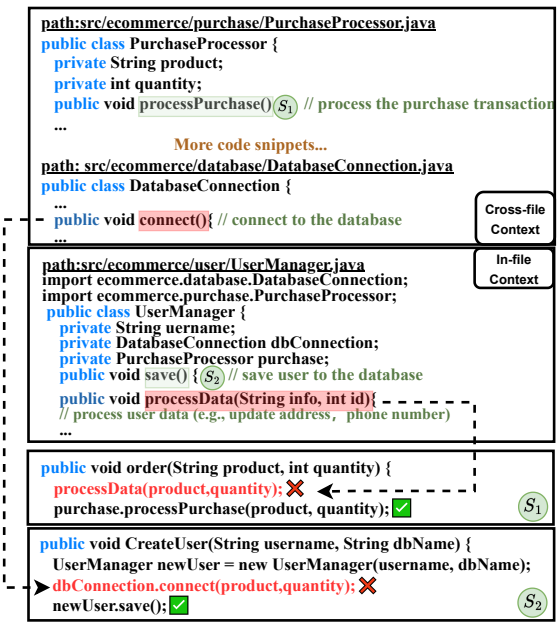

Figure 1: Situations of repo-level code completion.

As shown in Fig. 1 (s1), this example shows the LLM incorrectly focusing on method signatures in the local (i.e., in-file) context, while ignoring the broader global (i.e., cross-file) context. The LLM incorrectly invokes `processData(product, quantity)`, which fails to reflect the intended action of placing an order. The proper call is `processPurchase(product, quantity)` (*e.g.*, colored in green), which aligns with the intended purchase logic defined in the *PurchaseProcessor* class, imported from the cross-file context. **This situation highlights that the LLM can't correctly focus on local context or global context.**

As shown in Fig. 1 (s2), this example shows **an LLM's failure to deeply reason about the target cross-file object.** The LLM incorrectly calls `dbConnection.connect()` directly after instantiating *newUser* neglecting the initialization logic `newUser.save()` encapsulated within the *UserManager* class, which internally manages the database interaction.

Therefore, a key research question regarding repo-level code completion stands out, i.e., *how to improve the model's capability to utilize both cross-file and in-file context effectively?* It is a challenging task: (i) **Guiding LLMs to focus on the suitable context (e.g., in-file context or cross-file context or both) remains a non-trivial problem.** Different parts of the codebase depend on vastly different types of contextual information. Some completions can be resolved using local (in-file) scope, while others require a broader, cross-file perspective. LLMs still lack the capability to decide which type of context is relevant for a given completion task (Ding et al., 2023). (ii) **LLMs still struggle to reason deeply about the cross-file information due to complicated project dependencies and code structures.** From the repository perspective, code completion requires the LLM to have a global view that spans multiple files and modules. This means a deep understanding of how different components are architected and interact.

To tackle the above challenges, we propose a novel reinforcement learning (RL) approach for repo-level code completion, which is designed to improve the model's completion ability in different completion scenarios. To address the challenge of focusing on a suitable context, we leverage identifier-driven intent recognition to capture the completion intent. Specifically, we adopt the GRPO (Shao et al., 2024) method for autonomous learning and design an intent reward based on the matching degree between identifiers in the predicted and real code, guiding the LLM to focus on the correct context. To address the challenges of dependency resolution and cross-file reasoning, we propose a reward-driven completion learning utilizing a reward function for repo-level code completion, providing continuous and granular reward values based on the distance of the code editing as feedback for cross-file reasoning. To guide LLMs to effectively complete code that meets the intended functionality and adheres to the repository-level context, we design a selective-exploration strategy optimization that focuses on low-confidence, high-reward tokens in the model's multiple responses, reflecting divergent thought paths and motivating the model to explore underexplored yet more valuable code completion patterns.

In summary, our paper makes the following contributions: (1) A reinforcement learning method for repo-level code completion, which effectively addresses the issue of rewards in complex repository completion scenarios and improves cross-file reasoning, enabling the model to learn autonomously on real-world repository data. (2) The first deep research into LLMs' multi-scenario code completion ability, enhancing the model's ability to focus on a suitable repository context through intent recognition and selective exploration. (3) IntentCoder achieves state-of-the-art (SOTA) performance on repo-level code completion tasks with code LLM (7B parameters) on RepoBench, demonstrating strong and stable improvements.

2 Related Work

Large Language Models for Code. A wide range of LLMs (Fried et al., 2022; Chowdhery et al., 2023; Wang et al., 2023b; Zheng et al., 2023a; Christopoulou et al., 2022) tailored for code-related tasks, such as code comprehension and generation, have been developed and widely adopted in software engineering to enhance development efficiency (Chen et al., 2021; Liu et al., 2023). Some open-source models, Deepseek-Coder (Guo et al., 2025) and Qwen2.5-Coder (Hui et al., 2024), show strong code generation abilities. CodeLLMs have shown strong performance on benchmarks (Lai et al., 2023; Allamanis and Sutton, 2013; Lu et al., 2021; Raychev et al., 2016; Yu et al., 2024) for single-file code completion, where they can predict whole lines of code effectively. However, their performance in repo-level code completion remains limited, as LLMs struggle to capture the full repository context due to token size restrictions (Phan et al., 2024), focusing mainly on local context within individual files and neglecting cross-file dependencies (Liang et al., 2024).

Repository-level Code Completion. Repository-level code completion aims to provide code completion by leveraging the context of a repository. Given the large size of repositories and model context limitations, many approaches adopt the *retrieve-then-generate* paradigm (Agrawal et al., 2023; Cheng et al., 2024). Methods like A3-CodGen (Liao et al., 2023), CodePlan (Bairi et al.,

2024), and RepoFuse (Liang et al., 2024) use static code analysis for contextual integration to improve code completion. RepoCoder (Zhang et al., 2023) refines retrieval using feedback from generated code, while RLCoder (Wang et al., 2024) improves it with a stop signal mechanism. Despite advancements in context retrieval, effectively utilizing it remains challenging. FT2Ra (Guo et al., 2024) and CoCoMIC (Ding et al., 2022) integrate retrieved content, but require architectural changes, making them unsuitable for pre-trained LLMs. In contrast, aixcoder-v2 (Li et al., 2025) uses fine-tuning to assimilate long contexts for improved code completion. Unlike previous methods, we use reinforcement learning to enhance the LLM's performance in various scenarios.

3 Methodology

3.1 Task Definition and Motivation

Given an incomplete code snippet x (i.e., in-file context $C_{\text{in-file}}$) and its cross-file contexts C_{cross} in a specific repository, the task of repository-level code completion is to generate the completion y. More formally, the task's objective is to generate the code completion y by a model π_θ such that the conditional probability $P_\theta(y|C_{\text{in-file}}, C_{\text{cross}})$ is maximized. The definitions of in-file and cross-file contexts are in Appendix A.1, and the task motivation is in Appendix A.2.

3.2 Preliminary

Group Relative Policy Optimization (GRPO) is a method of reinforcement learning that explores the potential of LLMs to develop reasoning capabilities, focusing on their self-evolution through a pure RL process (Guo et al., 2025; Shao et al., 2024). GRPO is applied to estimate the advantages of language model generations by comparing responses within a query-specific group.

Specifically, in repo-level completion task, given an input $q = (C_{\text{in-file}}, C_{\text{cross}})$, GRPO samples a group of outputs $\{y_1, y_2, \ldots, y_G\}$ from the old policy model $\pi_{\theta_{\text{old}}}$, and then optimizes the policy model π_θ by maximizing the following objective:

$$\mathcal{J}_{\text{GRPO}}(\theta) = \mathbb{E}_{q\sim P(Q),\{y_i\}_{i=1}^{G}\sim\pi_{\theta_{\text{old}}}(y|q)} \frac{1}{G}\sum_{i=1}^{G} \min\left(\frac{\pi_\theta(y_i|q)}{\pi_{\theta_{\text{old}}}(y_i|q)}, 1-\epsilon, 1+\epsilon\right)A_i - \beta\mathcal{D}_{KL}(\pi_\theta\|\pi_{\text{ref}}), \quad (1)$$

where ϵ and β are hyper-parameters, $\mathcal{D}_{KL}$ is the penalty function of Kullback-Leibler divergence. π_{ref} is the reference model, which is initial by the policy model π_θ. A_i is the advantage for the i-th response, computed using a group of rewards $\{r_1, r_2, \ldots, r_G\}$ corresponding to the outputs within each group:

$$A_i = \frac{r_i - \text{mean}(r_1, r_2, \ldots, r_G)}{\text{std}(r_1, r_2, \ldots, r_G)} \quad (2)$$

3.3 Completion-Focused Reinforcement Learning

Even though LLM receives sufficient repository context information, it often fails to effectively leverage and understand it for accurate code completion. Inspired by reinforcement learning, which helps mitigate the reliance on empirical memory for code completion, we propose completion-focused reinforcement learning based on GRPO (Section 3.2) to enable the model to autonomously learn code completion in a repository.

Reward-Driven Completion Learning. Employing reward feedback based on result matching presents challenges in complex and diverse cross-file completion scenarios, as the model often fails to receive rewards due to the sparsity of signals when directly matching the ground truth code. To address this challenge, we propose reward-driven completion learning, which provides more precise and granular reward feedback in the complex repo-level code completion environment, enabling the model to receive effective reward signals that enhance its cross-file reasoning ability.

Specifically, given in-file context $C_{\text{in-file}}$ and the cross-file context $C_{\text{cross}} = \{C_i\}_{i=1}^{n}$ in repository, the i-th output y_i in a group is sampled by the old policy model $\pi_{\theta_{\text{old}}}$, as described by the following equations:

$$y_i = \pi_{\theta_{\text{old}}}(C_{\text{in-file}}, C_{\text{cross}}). \quad (3)$$

Following the prompt of repository-level completion in Fig. 4 (provided in Appendix A.3), we obtained a group of generated code $\{y_i\}_{i=1}^{G}$ through multiple samplings by Formula 3. Completion reward functions that evaluate the generated code are defined from the following perspectives: (1) **completion reward** provides a relative quality ranking of the generated code completions. The completion reward R_c for the i-th output y_i is defined as follows:

$$R_c = f_{\text{Edit}}(y_i, \hat{y}), \quad (4)$$

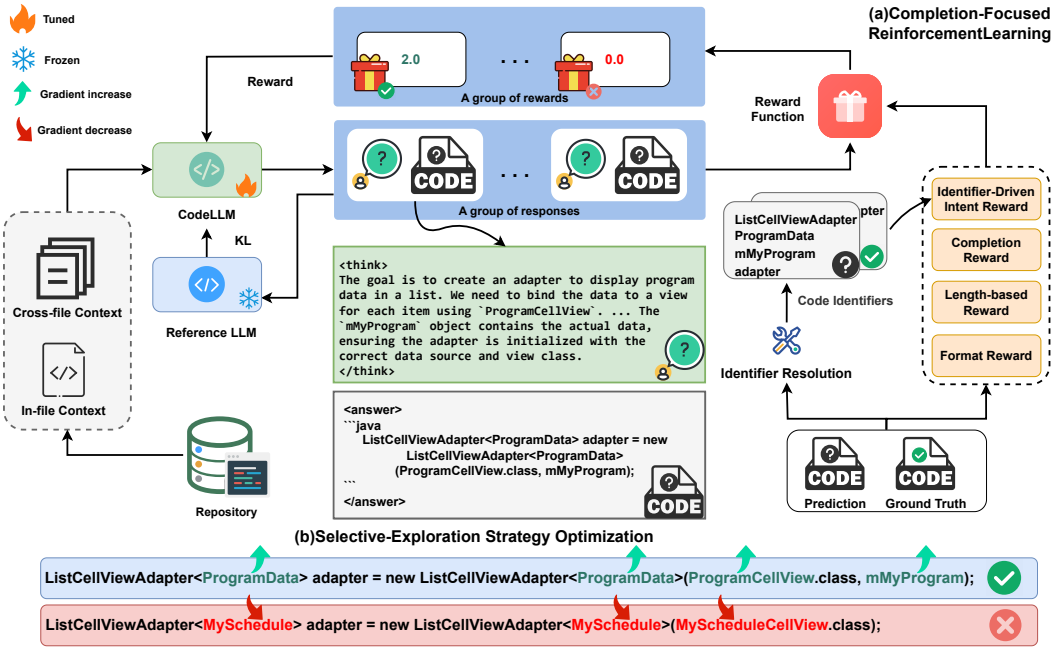

Figure 2: Overview of IntentCoder. (a) Illustration of the Completion-Focused RL process. (b) Illustration of the Selective-Exploration Strategy Optimization.

where $\hat{y}$ is the ground truth code, f_{Edit} is the Levenshtein distance algorithm (Yujian and Bo, 2007) to compare the character differences (*i.e.*, insertions, deletions, substitutions) between generated code and ground truth code. This reward provides fine-grained, continuous feedback (ranging from 0 to 1), ensuring the model receives quality feedback even without exact matches. (2) **Format reward** R_f ensures the model's output is evaluable by requiring the response to follow a specific format for reasoning and answers. A regular expression checks that these markers are present and correctly ordered. (3) **Length-based reward**: Extra code lines may receive spurious rewards, hindering accurate feedback and reducing training efficiency. To address this, we propose a Length-based reward R_L to regulate the number of generated lines for better alignment with code completion needs:

$$R_L(y_i) = \begin{cases} 1, & if\ len(y_i) = 1 \\ 0, & otherwise, \end{cases} \tag{5}$$

where len denotes the function that calculates the number of code lines. Based on these reward functions, we define the reward function R as follows:

$$R = \alpha_c * R_c + \alpha_f * R_{\text{f}} + \alpha_L * R_L, \tag{6}$$

where α_* is hyper-parameters. As shown in Algorithm 1, we use the reward function R to train the model π_θ, enabling the model to autonomously learn from repository data.

Algorithm 1 Completion-Focused Reinforcement Learning

Input: initial policy model π_θ^{init}, reward function R, dataset $\mathcal{D} = \{C_{\text{in-file}}, C_{\text{cross}}, \hat{y}\}$; hyperparameters μ
policy model $\pi_\theta \leftarrow \pi_\theta^{init}$
reference model $\pi_{\text{ref}} \leftarrow \pi_\theta$
for step = 1, ..., M **do**
  Sample a batch $\mathcal{D}_b$ from $\mathcal{D}$
  Update the old policy model $\pi_{\theta_{\text{old}}} \leftarrow \pi_\theta$
  $q \leftarrow \{C_{\text{in-file}}, C_{\text{cross}}\} \in \mathcal{D}_b$
  Sample $\{y_i\}_{i=1}^{G} \sim \pi_{\theta_{\text{old}}}(\cdot|C_{\text{in-file}}, C_{\text{cross}})$ for $q \in \mathcal{D}_b$
  $\{R_i\}_{i=1}^{G} \leftarrow R(y_i, \hat{y})$ for $y_i \in \{y_i\}_{i=1}^{G}$ (Equation 6)
  Compute A_i for $y_i \in \{y_i\}_{i=1}^{G}$ (Equation 2)
  for iteration = 1, ..., μ **do**
    Update π_θ by maximizing $\mathcal{J}_{\text{GRPO}}(\theta)$ (Equation 1)
  end for
end for
Output: π_θ

Identifier-Driven Intent Recognition. In real-world code completion scenarios, the performance of LLMs is often enhanced by utilizing cross-file context from the repository (Zhang et al., 2024; Eghbali and Pradel, 2024). However, the model struggles to focus on suitable contexts from in-file and cross-file contexts (Ding et al., 2023), resulting in incorrect code completions. To address this challenge, we propose Identifier-Driven Intent Recognition, which is designed to guide the LLM in identifying the developer's completion intent and understanding code contexts, thereby correctly focusing on suitable contexts.

Specifically, as shown in Fig. 4, we instruct the model's reasoning process in the prompt to focus on the user's intent of code completion (*e.g.*, col-

ored in blue), directing its reasoning towards the intent of completion in the reasoning proces. Given that identifiers contain semantic information from the code, we utilize an identifier-based reward function R_I to evaluate the degree of correctness in the model's intent recognition. As shown in Fig. 2, we parse the code and extract the identifiers (*e.g.*, variable, function, and class names) from the model's prediction y and ground truth code $\hat{y}$. The identifier-based reward function R_i is calculated as follows:

$$R_I = F_1(f_{\text{id}}(\pi_{\theta_{\text{old}}}(C_{\text{in-file}}, C_{\text{cross}})), f_{\text{id}}(\hat{y})), \quad (7)$$

where f_{id} extracts the sets of identifiers from code. F_1 is the harmonic mean of the precision and recall metric, which is used to evaluate the degree of intent consistency between the predicted and ground truth code. This approach enables the model to properly consider contextual information and adjust its focus based on the situation, thereby effectively leveraging the advantages of both in-file and cross-file contexts to improve performance.

To effectively coordinate the intent and completion rewards, we joint them as follows:

$$R_{ic} = (1 + R_I) * R_c, \quad (8)$$

$$R = \alpha_{ic} * R_{ic} + \alpha_f * R_{\text{f}} + \alpha_L * R_L, \quad (9)$$

where R is the overall reward function, α_* is hyperparameter. By combining these two reward signals, the model is encouraged to prioritize completing the code in a way that both fulfills the user's intended functionality and adheres to the repository-level context. When the edit distance between two generated codes is similar in a group of outputs $\{y_i\}_{i=1}^{G}$, the intent reward acts as a coefficient, allowing for a more accurate reward assignment and reducing the lack of distinction in reward values.

Selective-Exploration Strategy Optimization. The model's multiple responses to the same completion task reflect different thought paths, with some responses aligning in part as a consensus, while others show divergence on key points. Rewarding the key information where the model shows uncertainty can help cross-file reasoning and contextual attention in the correct contexts. During our reinforcement learning process, some key tokens within the group have a lower generation probability than the average of the tokens in the group of outputs, yet they might receive rewards higher than the average reward for the group. As shown in Fig. 2, this type of token (*e.g.*, colored in green) contains valuable information that significantly influences the reasoning and completion strategy. Based on the above observations, we propose a selective-exploration strategy optimization. The key idea is to encourage the model to focus more on tokens with lower confidence and higher reward, as these tokens represent the high-value and critical aspects of the solution that can be further improved for the model's completion learning.

Specifically, for the t-th token of the i-th output $y_{i,t}$ within the group, the confidence function P and value function V are as follows:

$$P(y_{i,t}) = \log \pi_\theta(y_{i,t}|q, y_{<t}) - \frac{1}{N}\sum_{j=1}^{G}\sum_{t=1}^{L_i} \log \pi_\theta(y_{i,t}|q, y_{<t}),$$

$$V(y_{i,t}) = A(y_{i,t}) - \frac{1}{G}\sum_{j=1}^{G} A(y_j), \quad (10)$$

where N is the total number of tokens in a group of outputs, L_i is the number of tokens in the i-th output. Then the tokens that have high value and low confidence are selected as follows:

$$T_s = \{y_{i,t} \mid P(y_{i,t}) < 0, V(y_{i,t}) > 0\}, \quad (11)$$

where T_{select} is the set of selected tokens. To simplify the expression and enhance readability, we define the function $Adv(y_{i,t})$ as follows:

$$Adv(\cdot) = \min\left(\frac{\pi_\theta(y_{i,t}|q, y_{i,<t})}{\pi_{\theta_{\text{old}}}(y_{i,t}|q, y_{i,<t})}, 1-\epsilon, 1+\epsilon\right)A_i. \quad (12)$$

Based on the function Adv, the training of the model π_θ for the position of token $y_{i,t}$ is defined as maximizing the following objective:

$$\mathcal{J}(y_{i,t}) = \begin{cases} Adv(y_{i,t}) - \beta\mathcal{D}_{KL}(\pi_\theta \| \pi_{\text{ref}}), & y_{i,t} \notin T_s \\ Adv(y_{i,t}), & y_{i,t} \in T_s, \end{cases} \quad (13)$$

where token $y_{i,t} \in T_{\text{selected}}$ removes the KL penalty compared to other tokens. By applying fine-grained gradient updates to tokens in different areas, we adjust the model's attention away from tokens where it already has high confidence and shift it towards those tokens that have the potential to improve the overall output. This approach encourages the model to gain a deeper understanding of code completion by focusing on divergent contextual areas and exploring valuable code completion patterns.

4 Experiments

4.1 Dataset

To study repository-level code completion in the context of in-file and cross-file scenarios, we construct the **ICRepo** dataset (**I**n-file & **C**ross-file

Repository-level Completion) for model training. We collect 32K repository-level code completion examples from the RepoBench-C and RepoBench-P datasets (Liu et al., 2023), with an equal number of Python and Java examples. We contain three completion scenarios as follows: **Cross-File-First (CF-F)**, **Cross-File-Random (CF-R)**, **In-File (IF)**. Overall statistics of the dataset are given in Table 1. The training dataset encompasses a variety of real-world code completion scenarios, ensuring the model is exposed to diverse challenges. By incorporating both in-file and cross-file contexts, the dataset covers practical code completion situations that developers frequently encounter in actual projects. For the test dataset, we use Repobench-Java and Repobench-Python (Liu et al., 2023). Further Details are provided in Appendix A.4.

Language	Sets	Num.	Completion position		
			IF	CF-F	CF-R
Java&Python	train	32,000	10,100	11,300	10,600
	val	1,000	300	400	300
	test	6,000	2,000	2,000	2,000

Table 1: Dataset Statistics of ICRepo.

4.2 Baselines

We consider the following open-source Code LLMs for comparison, which are widely used in code completion tasks. To simplify the model names, we have removed the "instruct" suffix in the following introductions: **StarCoder2** (Lozhkov et al., 2024): StarCoder2-15B and StarCoder2-7B. **DeepSeek-Coder** (Guo et al., 2025): Deepseek-Coder-6.7B and DeepSeek-Coder-33B. **CodeLlama** (Roziere et al., 2023): CodeLlama-7B and CodeLlama-34B. **Qwen2.5-Coder** (Hui et al., 2024): Qwen2.5-Coder-7B and Qwen2.5-Coder-33B. **aixcoder-7B-v2** (Li et al., 2025): aixcoder-7B-v2 is currently the SOTA 7B model in repository-level code completion. Then we compare our method with the following methods in the training model for repository-level code completion: **fine-tuning** (Hu et al., 2022) and **GRPO** (Guo et al., 2025). Further Details are provided in Appendix A.5.

4.3 Implementation Details

We train IntentCoder based on Deepseek-Coder-6.7B and Qwen2.5-Coder-7B, respectively. For each prompt of training data, the old policy model $\pi_{\theta_{\text{old}}}$ generates 6 outputs in a group using a temperature of 0.7 and top-p sampling with a value of 1.0. To improve training efficiency, we use LoRA (Hu et al., 2022), vllm (Kwon et al., 2023), and deepspeed (Rasley et al., 2020) for train the model π_θ. Further details are provided in the Appendix A.6.

4.4 Evaluation Metrics

Following the previous work (Liu et al., 2023; Eghbali and Pradel, 2024; Zhang et al., 2023; Jiang et al., 2024b), we report performance in two main categories to evaluate the performance of codeLLMs: **Code Match**: To evaluate the accuracy of the predicted code lines, we use **Exact Match (EM)** and **Edit Similarity (ES)** (Lcvenshtcin, 1966). **Identifier Match** (Ding et al., 2022, 2023): To evaluate the model's ability to understand the completion intent and predict the correct APIs, we use the following metrics about Identifier: **Exact Match (EM)**,**Precision**,**Recall** and **F1**. Further details can be found in Appendix A.7.

4.5 Experimental Results

Repo-level Code Completion Evaluation. To evaluate the effectiveness of IntentCoder, we apply our training method to Qwen2.5-Coder-7B and DeepSeek-Coder-6.7B, and then compare it with nine Code LLMs and two strong training methods for LLMs. Table 2 shows the experimental results of our model and other baselines. It is obvious that: (1) Regarding the comparison with other models, IntentCoder outperforms all strong baselines by a large margin in both code match and identifier match, surpassing even larger models. For example, IntentCoder$_{\text{Qwen}}$ achieves 42.47% and 35.93% in Code EM for Java and Python, surpassing the next best model (*i.e.*, Qwen2.5-Coder-32B) by a significant margin of 6.53% and 5.87%. Similarly, in terms of F1 in Identifier Match, IntentCoder$_{\text{Qwen}}$ outperforms the next best model (*i.e.*, Qwen2.5-Coder-32B) by a significant margin of 5.97% and 4.97%. Compared to models with the same parameters in Code EM, IntentCoder$_{\text{Qwen}}$ outperforms the current SOTA model (*i.e.*, aixcoder-7B-v2) by a significant margin of 8.53% in Java and 12.07% in Python, which clearly demonstrates our model's code completion capability. (2) Regarding the comparison with other methods, our approach demonstrates substantial improvements across all evaluation metrics over other training methods on two main strong baselines (*e.g.*, Qwen2.5-Coder-7B, DeepSeek-Coder-6.7B). Specifically, IntentCoder$_{\text{DeepSeek}}$ outperforms DeepSeek-Coder-6.7B by up to 11.40% and 20.30% in Code EM for Java and Python. Similarly, IntentCoder$_{\text{Qwen}}$

Model	RepoBench-Java						RepoBench-Python					
	Code Match		Identifier Match				Code Match		Identifier Match			
	EM	ES	EM	Precision	Recall	F1	EM	ES	EM	Precision	Recall	F1
StarCoder2-7B	12.20	29.78	15.37	24.64	28.19	25.09	1.80	14.47	3.83	7.72	13.25	8.40
CodeLlama-7B	12.40	48.50	15.90	35.60	41.60	36.46	4.47	25.56	6.37	18.26	26.68	18.84
aixcoder-7B-v2	33.07	68.97	38.70	60.57	65.97	61.07	23.20	59.94	29.63	54.70	62.84	55.41
Starcoder2-15B	12.20	28.59	15.67	24.51	27.77	24.92	1.83	13.70	4.07	7.66	13.05	8.34
CodeLlama-34B	12.13	45.39	15.43	33.59	39.11	34.01	4.40	26.02	6.70	18.97	27.55	19.07
DeepSeek-Coder-33B	26.70	62.03	31.53	53.11	57.99	53.39	14.03	45.00	18.57	37.81	45.49	38.30
Qwen2.5-Coder-32B	35.07	71.13	41.10	63.12	65.28	62.56	29.40	63.82	35.40	57.52	61.69	57.43
DeepSeek-Coder-6.7B	30.20	66.28	35.97	57.67	62.81	57.94	14.97	43.87	19.20	38.13	47.20	38.94
DeepSeek-Coder-6.7B$_{\text{fine-tuning}}$	38.93	74.37	44.77	66.23	66.17	65.22	31.27	66.36	37.53	59.44	58.76	57.66
DeepSeek-Coder-6.7B$_{\text{GRPO}}$	35.47	71.29	41.57	63.95	68.13	63.83	34.70	68.82	40.93	62.87	60.64	60.39
IntentCoder$_{\text{DeepSeek}}$	**41.60**	**77.64**	**47.73**	**69.74**	**69.18**	**68.53**	**35.27**	**71.12**	**42.10**	**64.41**	**62.98**	**62.39**
Qwen2.5-Coder-7B	26.80	62.77	32.70	54.74	57.06	53.81	19.97	52.39	25.20	46.55	52.45	46.51
Qwen2.5-Coder-7B$_{\text{fine-tuning}}$	37.43	74.77	43.50	65.93	65.78	64.88	31.53	66.61	37.53	59.59	58.69	57.81
Qwen2.5-Coder-7B$_{\text{GRPO}}$	37.50	73.49	43.60	65.87	67.58	65.02	35.13	68.71	41.20	62.21	60.10	59.86
IntentCoder$_{\text{Qwen}}$	**42.47**	**77.52**	**48.57**	**69.95**	**69.60**	**68.75**	**35.93**	**71.58**	**42.70**	**65.57**	**63.80**	**63.37**

Table 2: Evaluation results on the RepoBench-java/Python dataset. All results in the table are reported in percentage (%). The best method is in boldface, and the second best method is underlined for each metric.

outperforms Qwen2.5-Coder-7B by up to 15.67% and 15.96% . **Overll, IntentCoder shows substantial and stable improvements compared to other models and methods.**

Model	Code Match		Identifier Match			
	EM	ES	EM	Precision	Recall	F1
IntentCoder	**35.93**	**71.58**	**42.70**	**65.57**	**63.80**	**63.37**
w/o IDIR	34.60	69.98	41.27	63.74	63.10	62.05
w/o RDCL	34.30	68.97	40.20	62.24	60.87	60.25
w/o CFCL	34.47	67.73	40.70	62.06	59.61	59.45
w/o SESO	34.47	70.43	40.97	64.58	62.99	62.39
w/o All	19.97	52.39	25.20	46.55	52.45	46.51

Table 3: Results of Ablation study on the RepoBench-Python dataset in Qwen2.5-Coder-7b.

Contributions of Each Component. In this RQ, we conduct an ablation study on IntentCoder$_{\text{Qwen}}$ to assess the contribution of different techniques by removing key components (*i.e.*, identifier-driven intent recognition (IDIR), reward-driven completion learning (RDCL), completion-focused RL (CFRL), selective-exploration strategy optimization (SESO)) of our approach separately. For IDIR and RDCL components, we separately remove their the design of the reward function and retain the basic RL process (*i.e.*, GRPO). Similarly, for CFCL, we remove all reward design (*i.e.*, IDIR, RDCL), retain the intention prompt and adopt Code EM as a substitute for reward. As illustrated in Table 3, The experimental results show that: (1) No matter which component we drop, it consistently hurts the overall performance of IntentCoder across all metrics, signaling the importance and effectiveness of all four components. When all the components were removed, the performance of the model significantly declined. (2) When CFCL is removed, the code match and identifier match metrics, except for the EM metric, show a more significant drop than when removing IDIR and RDCL separately. This justifies the importance and necessity of CFCL in effectively coordinating the intent and completion rewards, which both fulfill the user's intended functionality and adhere to the repository-level context. Notably, EM is slightly higher than removing any one of them (*e.g.*, colored in gray), since we use the exact code match as an alternative to the reward function. Similarly, the ablation study on Java is provided in Appendix Table 6.

Robustness Evaluation. To evaluate the robustness of our training method, we assess the influence of different training methods (*i.e.*, fine-tuning, GRPO) on reducing cross-file context-induced failures (*i.e.*, fail) and improving the accuracy of completion (*i.e.*, pass). As shown in Table 4, compared to the baseline models, our training method significantly reduces cross-file context-induced failures and improves completion accuracy, thereby improving overall performance. For instance, our approach has reduced the number of failures of DeepSeek-Coder-6.7b on the Python dataset from -130 to -41 and increased the number of pass from 189 to 709, representing an overall improvement of 668. Compared with other methods, our approach reduces the failure rate comparable to GRPO and significantly improves the pass rate compared to

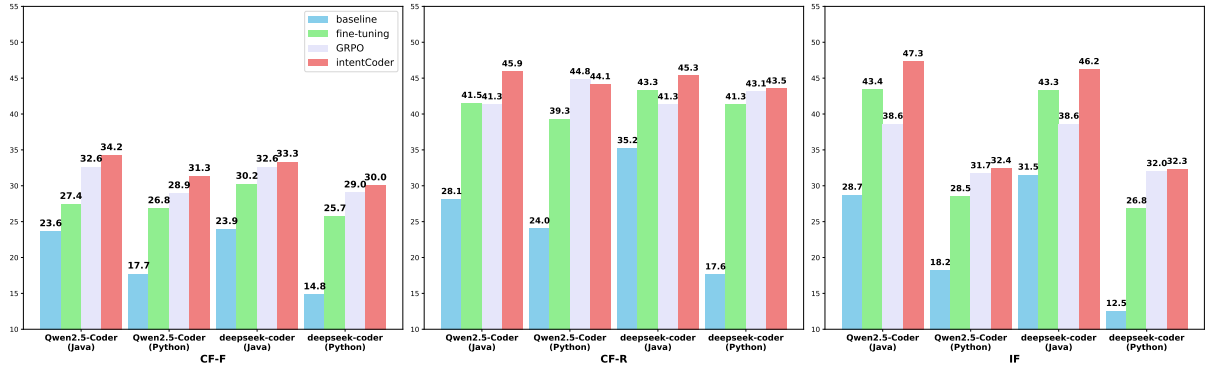

Figure 3: Performance of IntentCoder in Various Completion Scenarios (CF-F,CF-R,IF).

the other two methods, thereby achieve significant overall improvement. This justifies the importance and robustness of our method in real-world code completion scenarios, effectively handling and understanding in-file and cross-file contexts.

Method	Java			Python		
	fail	pass	overall	fail	pass	overall
DeepSeek-Coder-6.7B	-131	+351	+220	-130	+189	+59
fine-tuning	-117	+599	+482	-61	+609	+548
GRPO	**-78**	+456	+378	**-40**	+691	+651
IntentCoder$_{\text{deepseek}}$	-83	**+645**	**+562**	-41	**+709**	**+668**
Qwen2.5-Coder-7B	-128	+301	+173	-138	+225	+87
fine-tuning	-105	+597	+492	-113	+547	+434
GRPO	**-54**	+548	+494	-62	+604	+542
IntentCoder$_{\text{Qwen}}$	-58	**+701**	**+643**	**-60**	**+626**	**+566**

Table 4: Result of Exact Match (EM) count.

Various Scenarios Evaluation. To further evaluate the effectiveness of IntentCoder in various completion scenarios, we compare our approach with other methods (*i.e.*, fine-tuning, GRPO) using the Code EM metric. As shown in Fig.3, in all completion scenarios (CF-F, CF-R, IF) that illustated in Appendix A.4.1, our method generally outperforms other training methods, although there is one instance where it performs slightly worse. The experimental results show that: (1) In the IF setting, IntentCoder performs significantly better than other methods, indicating that IntentCoder is not reliant on guessing based on prior cross-file information and effectively handles in-file contexts. (2) Similarly, in the CF-F setting, IntentCoder outperforms other methods consistently, demonstrating that it does not incorrectly depend on in-file context or previous cross-file information, and accurately handles cross-file dependencies, even when encountered for the first time. Overall, compared to the baseline, IntentCoder achieves stable improvement and effectively handles different completion scenarios.

Runtime latency and Case Study. To evaluate the infulence of runtime latency, We conducted experiments on a test set of 3000 samples on IntentCoder using the vLLM inference library. Since the test dataset RepoBench-Java/Python enforces input token limit of 2048, the size of the context fed into our IntentCoder remained below this threshold (on average 1121 tokens), the input latency is under 238 ms. The maximum output token length of the <think> block is 119, with an average length of 20 tokens. The additional latency is small, with an average delay of less than 25 ms and a maximum delay not exceeding 145 ms. We believe the small increase in latency is acceptable because the difference is imperceptible to humans and the accuracy of code completion is improved. To illustrate the performance of IntentCoder compared to other models, we have included a case study in the Appendix A.8.

5 Conclusion

This research aims to improve the Code LLM's capability of repo-level code completion by enhancing their capabilities in handling both in-file and cross-file contexts. To perform this task, we propose a novel training approach designed to improve LLMs' understanding of complex repo-level code by incorporating reward-driven learning for better cross-file reasoning and intent recognition to focus on suitable contexts from in-file and cross-file contexts. Selective-exploration strategy optimization guides the model in exploring more valuable code completion patterns, resulting in more accurate and context-aware code suggestions. Experimental results demonstrate that the effectiveness of IntentCoder . We hope our work advances LLMs to handle real-world repo-level challenges and provides valuable insights into improving code LLMs for software engineering development.

6 Limitations.

Several limitations are concerned with our work. Firstly, we do not utilize execution-based rewards due to the complexities involved in constructing executable environments, which are difficult to scale for large datasets. Additionally, the time-intensive nature of software building and testing renders this approach impractical for model training. Although execution-based rewards represent a valuable research avenue, our focus is on rewards derived from edit similarity and identifier matching. This design is more flexible, scalable, and efficient, enabling practical application across large datasets, even when execution is not feasible, and supporting sparse rewards.

Secondly, in this work, we intentionally focused only on strengthening the generation side. i.e., improving code completion results by effectively using cross-file and in-file context. IntentCoder can naturally be integrated into any retrieval-augmented pipeline as the generator component. However, we believe that research on how to enable LLMs to search and select contexts in a repository within an acceptable time cost and context input limitation is an interesting topic for our future work.

Thirdly, while larger and more capable LLMs may yield better reasoning results during RL training, they come with higher computational costs, such as increased GPU memory usage and longer training times. Due to resource constraints and our goal of supporting community research, we only trained on 7B. We have released our replication package to enable others to replicate our work and verify their ideas.

References

Lakshya A Agrawal, Aditya Kanade, Navin Goyal, Shuvendu Lahiri, and Sriram Rajamani. 2023. Monitor-guided decoding of code lms with static analysis of repository context. *Advances in Neural Information Processing Systems*, 36:32270–32298.

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)*, pages 207–216. IEEE.

Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, and 1 others. 2022. Multi-lingual evaluation of code generation models. *arXiv preprint arXiv:2210.14868*.

Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D C, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, Balasubramanyan Ashok, and Shashank Shet. 2024. Codeplan: Repository-level coding using llms and planning. *Proceedings of the ACM on Software Engineering*, 1(FSE):675–698.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Wei Cheng, Yuhan Wu, and Wei Hu. 2024. Dataflow-guided retrieval augmentation for repository-level code completion. *arXiv preprint arXiv:2405.19782*.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, and 1 others. 2023. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113.

Fenia Christopoulou, Gerasimos Lampouras, Milan Gritta, Guchun Zhang, Yinpeng Guo, Zhongqi Li, Qi Zhang, Meng Xiao, Bo Shen, Lin Li, and 1 others. 2022. Pangu-coder: Program synthesis with function-level language modeling. *arXiv preprint arXiv:2207.11280*.

Yangruibo Ding, Zijian Wang, Wasi Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and 1 others. 2023. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. *Advances in Neural Information Processing Systems*, 36:46701–46723.

Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2022. Cocomic: Code completion by jointly modeling in-file and cross-file context. *arXiv preprint arXiv:2212.10007*.

Aryaz Eghbali and Michael Pradel. 2024. Dehallucinator: Iterative grounding for llm-based code completion. *arXiv preprint arXiv:2401.01701*.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, and 1 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

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. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 313–324.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, and 1 others. 2022. Lora: Low-rank adaptation of large language models. *ICLR*, 1(2):3.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, and 1 others. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

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*, pages 401–412.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024a. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*.

Siyuan Jiang, Jia Li, He Zong, Huanyu Liu, Hao Zhu, Shukai Hu, Erlu Li, Jiazheng Ding, Yu Han, Wei Ning, and 1 others. 2024b. aixcoder-7b: A lightweight and effective large language model for code completion. *arXiv e-prints*, pages arXiv–2410.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pages 18319–18345. PMLR.

VI Lcvenshtcin. 1966. Binary coors capable or 'correcting deletions, insertions, and reversals. In *Soviet physics-doklady*, volume 10.

Jia Li, Hao Zhu, Huanyu Liu, Xianjie Shi, He Zong, Yihong Dong, Kechi Zhang, Siyuan Jiang, Zhi Jin, and Ge Li. 2025. aixcoder-7b-v2: Training llms to fully utilize the long context in repository-level code completion. *arXiv preprint arXiv:2503.15301*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, and 1 others. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Ming Liang, Xiaoheng Xie, Gehao Zhang, Xunjin Zheng, Peng Di, Hongwei Chen, Chengpeng Wang, Gang Fan, and 1 others. 2024. Repofuse: Repository-level code completion with fused dual context. *arXiv preprint arXiv:2402.14323*.

Dianshu Liao, Shidong Pan, Qing Huang, Xiaoxue Ren, Zhenchang Xing, Huan Jin, and Qinying Li. 2023. Context-aware code generation framework for code repositories: Local, global, and third-party library awareness. *CoRR*.

Fang Liu, Zhiyi Fu, Ge Li, Zhi Jin, Hui Liu, Yiyang Hao, and Li Zhang. 2024. Non-autoregressive line-level code completion. *ACM Transactions on Software Engineering and Methodology*, 33(5):1–34.

Tianyang Liu, Canwen Xu, and Julian McAuley. 2023. Repobench: Benchmarking repository-level code auto-completion systems. *arXiv preprint arXiv:2306.03091*.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, and 1 others. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, and 1 others. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*.

Huy Nhat Phan, Hoang Nhat Phan, Tien N Nguyen, and Nghi DQ Bui. 2024. Repohyper: Better context retrieval is all you need for repository-level code completion. *CoRR*.

Shanghaoran Quan, Jiaxi Yang, Bowen Yu, Bo Zheng, Dayiheng Liu, An Yang, Xuancheng Ren, Bofei Gao, Yibo Miao, Yunlong Feng, and 1 others. 2025. Codeelo: Benchmarking competition-level code generation of llms with human-comparable elo ratings. *arXiv preprint arXiv:2501.01257*.

Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 3505–3506.

Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016. Probabilistic model for code with decision trees. *ACM SIGPLAN Notices*, 51(10):731–747.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, and 1 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, and 1 others. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

Chaozheng Wang, Junhao Hu, Cuiyun Gao, Yu Jin, Tao Xie, Hailiang Huang, Zhenyu Lei, and Yuetang Deng. 2023a. How practitioners expect code completion? In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1294–1306.

Yanlin Wang, Yanli Wang, Daya Guo, Jiachi Chen, Ruikai Zhang, Yuchi Ma, and Zibin Zheng. 2024. Rlcoder: Reinforcement learning for repository-level code completion. *arXiv preprint arXiv:2407.19487*.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023b. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*.

Zhaojian Yu, Yilun Zhao, Arman Cohan, and Xiao-Ping Zhang. 2024. Humaneval pro and mbpp pro: Evaluating large language models on self-invoking code generation. *arXiv preprint arXiv:2412.21199*.

Li Yujian and Liu Bo. 2007. A normalized levenshtein distance metric. *IEEE transactions on pattern analysis and machine intelligence*, 29(6):1091–1095.

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 preprint arXiv:2303.12570*.

Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. 2024. Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. *arXiv preprint arXiv:2401.07339*.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, and 1 others. 2023a. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5673–5684.

Zibin Zheng, Kaiwen Ning, Yanlin Wang, Jingwen Zhang, Dewu Zheng, Mingxi Ye, and Jiachi Chen. 2023b. A survey of large language models for code: Evolution, benchmarking, and future trends. *arXiv preprint arXiv:2311.10372*.

Wen Zhou, Seohyun Kim, Vijayaraghavan Murali, and Gareth Ari Aye. 2022. Improving code autocompletion with transfer learning. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, pages 161–162.

A Appendix

A.1 Definition

In repository-level code completion, in-file contexts and cross-file contexts refer to different types of contextual information that help a model to predict and complete code snippets:

(1) **In-file Context:** Following the previous work (Allamanis and Sutton, 2013; Chen et al., 2021; Athiwaratkun et al., 2022), in-file contexts refer to the information contained within the prefix of the current file itself, specifically the code that precedes the ground truth code in the incomplete code snippet. In-file context provides the immediate surrounding code before the incomplete snippet, which typically helps the model understand the structure, syntax, and logic of the code. This context is crucial for making predictions when code completion can be inferred solely from the current file's contents, without the need for additional external context.

(2) **Cross-file Context:** Cross-file Context refers to the information contained in external files within the repository upon which the current file depends. Such information can include functions, classes, or libraries imported into the current file. This type of context is extracted by analyzing dependencies or references to other files within the repository (Ding et al., 2022), providing a broader scope of information for code completion. Leveraging the cross-file context is essential when the in-file context is insufficient for completing the code.

Both types of In-file and Cross-file contexts are essential for repository-level code completion, allowing the model to make more informed predictions and generate accurate completions.

A.2 Motivation

In this study, we assume that if LLMs can effectively select between global context and local context, the repository-level code completion performance can be significantly boosted. To further verify our assumption, we conduct a preliminary experiment. In detail, we evaluate the performance of utilizing cross-context versus not using it for code completion in the RepoBench datasets (see details about RepoBench in Section 4.1) As shown in Table 5, "In-filet" column refers to correct completion using only the context from the current file. While "Both" column denotes correct completions obtained using both in-file and cross-file context. During the process of incorporating cross-files, it

model	Java			Python		
	In-file	→	Both	In-file	→	Both
CodeLlama-7B	290	-143|+227	374	134	-63|+100	174
StarCoder2-7B	31	-143|+192	80	55	-30|+29	54
DeepSeek-Coder-6.7B	686	-131|+351	906	390	-130|+189	449
Qwen2.5-Coder-7B	631	-128|+301	804	489	-138|+225	576
DeepSeek-Coder-33B	541	-150|+410	801	373	-166|+214	421
Qwen2.5-Coder-32B	844	-99|+307	1,052	771	-126|+237	882

Table 5: Results of Exact Match (EM) Count Code Completion.

introduces context-induced failures (*i.e.*, the red numbers). For example, CodeLlama-7B shows that the number of correct completions increases from 290 to 374 after using cross-file context, but 143 predictions that were originally correct under in-file context turn incorrect when additional cross-file context is included. This reveals a critical challenge: while cross-file context improves overall correctness, it simultaneously destabilizes previously correct predictions. It indicates that LLMs still lack reasoning capabilities for cross-file context and motivates us to explore how to guide LLMs to maximize cross-file benefits while mitigating context-induced failures. The experimental results indicate that while introducing cross-file context improves overall performance, it also harms the model's original performance in in-file scenarios.

A.3 Prompt

As show in Fig. 4, we use this prompt as the LLMs' input template.

• **Instruction**: You are a helpful assistant tasked with completing the next line of a repository-level code file. You are provided with an incomplete code file and other code snippets from the same repository. You first think about the reasoning process as an internal monologue that infers the user's intent of completion. Then, you should provide the user with the next line of the incomplete code.
Please respond in the following format:
<think>. . .</think>
<answer>``` . . .```</answer>

Repository name:{REPOSITORY _NAME}
• **Cross-file context** C_{cross}:
the path of cross-file context C_1
cross-file context C_1
. . .
the path of cross-file context C_n
cross-file context C_n
• **In-file context** $C_{\text{in-file}}$:
the path of incomplete code snippet x
incomplete code snippet x

Figure 4: the prompt of repository-level completion.

A.4 Dataset

A.4.1 Completion Scenarios

We contain three completion scenarios as follows:

Cross-File-First (CF-F): In this setting, the position of completion is the first appearance of a cross-file line within a file and ensuring there is no identical or similar cross-file invocation for the model to reference.

Cross-File-Random (CF-R): In this setting, the position of completion is a random and non-first occurrence of a cross-file line.

In-File (IF): In this setting, the position of completion is an in-file line that does not involve any cross-file modules. This setting serves as a robustness test to ensure that the incorporation of cross-file context does not greatly affect the accuracy of predictions.

A.4.2 Dataset Construction

We collect 32K repository-level code completion examples from the RepoBench-C and RepoBench-P datasets (Liu et al., 2023), with an equal number of Python and Java examples. The original RepoBench dataset supports Python&Java with three tasks (Retrieval/Completion/Pipeline). Our dataset is derived from Completion(RepoBench-C) and Pipeline(RepoBench-P) as they directly correspond to in-file and cross-file scenarios. Specifically, the context fed into our IntentCoder includes three parts: Pre-target-line crossfile context, Pre-target-line infile context, and Post-target-line crossfile noise. The Pre-target-line crossfile context refers to import modules that appear before the target prediction line. The pre-target-line infile context refers to code content from the beginning of the current file up to the target line. The post-target-line crossfile noise refers to import modules that occur after the target line, which are regarded as noise since we can only use context before the target line for code completion task. For the test dataset, We use Repobench-Java and Repobench-Python (Liu

et al., 2023), each containing 3,000 samples, with 1,000 samples for each of the CF-F, CF-R, and IF settings. Regarding context size, following the RepoBnech, Our dataset is designed for LLMs with a token limit of 2048. The average context size fed into the model is 1346 tokens for training, 1399 tokens for evaluation, and 1121 tokens for testing.

A.5 Baselines

We consider the following open-source CodeLLMs for comparison, which are widely used in code completion tasks. To simplify the model names, we have removed the "instruct" suffix in the following introductions (*e.g.*, Qwen2.5-coder, CodeLlama, Deepseek-Coder):

StarCoder2 (Lozhkov et al., 2024): The StarCoder2-15B is a 15 billion parameter model trained on 600+ programming languages using 4.3 trillion tokens, while the StarCoder2-7B is a 7 billion parameter model trained on 17 programming languages with 3.7 trillion tokens.

DeepSeek-Coder (Guo et al., 2025): Deepseek-Coder-6.7B is a 6.7 billion parameter model initialized from Deepseek-Coder-6.7b-base and fine-tuned on 2B tokens of instruction data. DeepSeek-Coder-33B is the largest variant of the DeepSeek-Coder series.

CodeLlama (Roziere et al., 2023): CodeLlama-7B and CodeLlama-34B are designed to be safer for use in code assistants and generation applications, with a focus on general code synthesis, understanding, and instruction-following.

Qwen2.5-Coder (Hui et al., 2024): Qwen2.5-Coder-7B and Qwen2.5-Coder-33B have performed in code generation, reasoning, and fixing, with the latter scaling up training tokens to 5.5 trillion. Qwen2.5-Coder-33B is the SOTA open-source CodeLLM, offering coding abilities comparable to GPT-4o and providing a more comprehensive foundation for real-world applications.

aixcoder-7B-v2 (Li et al., 2025): aixcoder-7B-v2 is currently the SOTA 7B model in repository-level code completion, fine-tuned based on aiXcoder-7B to utilize information within long contexts for code completion tasks.

Then we compare our method with the following methods in the training model for repository-level code completion:

fine-tuning (Hu et al., 2022): A method where pre-trained models are further trained on a specific task or dataset, refining the model's performance for the target application. We use Low-Rank Adaptation (LoRA) as the specific method.

GRPO (Guo et al., 2025): Group Relative Policy Optimization (GRPO), which is a method of reinforcement learning. We utilizes reinforcement learning by exact match as a reward to improve code prediction accuracy for repository-level code completion.

A.6 Implementation Details

The rank and alpha values of LoRA (Hu et al., 2022) are set to 16 and 32, respectively. We use a learning rate of 1×10^{-4} with a linear scheduler and warm-up, with the warm-up ratio set to 0.1. The model π_θ is trained on 4 NVIDIA A800 (80GB) GPUs for training and 1 GPU for inference simultaneously, which takes 10 hours. The batch size is 192. The hyperparameters μ, ϵ, and β are set to 1, 0.2, and 1×10^{-3}, respectively. The hyperparameters for reward weights, α_{ic}, α_f, and α_L, are set to 1.0, 0.2, and 0.1, respectively. We provide well-structured code interface, where reward functions can easily be configured and used. Other Details about the implementation are provided in our released code.

A.7 Evaluation Metrics

Following the previous work (Liu et al., 2023; Eghbali and Pradel, 2024; Zhang et al., 2023; Jiang et al., 2024b), we report performance in two main categories to evaluate the performance of codeLLMs:

Code Match: To evaluate the accuracy of the predicted code lines, we use **Exact Match (EM)** and **Edit Similarity (ES)** (Lcvenshtcin, 1966), which respectively represent the proportion of exact matches and the degree of similarity to the ground truth.

Identifier Match (Ding et al., 2022, 2023): To evaluate the model's ability to understand the completion intent and predict the correct application programming interfaces (APIs), we use the following metrics:(1) **Exact Match (EM)**: EM represents the proportion of exact matches to the ground truths in code identifiers. (2) **Precision**: Precision measures the proportion of correctly predicted code identifiers out of all predicted identifiers. (3) **Recall**: Recall measures the proportion of correctly predicted code identifiers out of all actual code identifiers in the ground truth. (4) **F1**: F1 is the harmonic mean of precision and recall metrics, providing a balanced metric for the model's performance.

Figure 5: The case study for IntentCodercompares it with other Code LLMs.

Model	RepoBench-Java						RepoBench-Python					
	Code Match		Identifier Match				Code Match		Identifier Match			
	EM	ES	EM	Precision	Recall	F1	EM	ES	EM	Precision	Recall	F1
IntentCoder$_{Qwen}$	**42.47**	**77.52**	**48.57**	**69.95**	**69.60**	**68.75**	**35.93**	**71.58**	**42.70**	**65.57**	**63.80**	**63.37**
w/o Identifier-Driven Intent Recognition	40.27	76.77	46.47	68.73	68.62	67.67	34.6	69.98	41.27	63.74	63.1	62.05
w/o Reward-driven Completion Learning	40.80	76.94	47.67	69.56	68.49	68.03	34.30	68.97	40.20	62.24	60.87	60.25
w/o Completion-focused RL	40.90	76.13	47.07	68.49	67.62	66.95	34.47	67.73	40.70	62.06	59.61	59.45
w/o Selective-Exploration Optimization	40.87	76.75	47.07	68.92	68.41	67.69	34.47	70.43	40.97	64.58	62.99	62.39
w/o ALL	26.80	62.77	32.70	54.74	57.06	53.81	19.97	52.39	25.20	46.55	52.45	46.51

Table 6: Results of Ablation study on the RepoBench-java and RepoBench-Python dataset. All results in the table are reported in percentage (%).

A.8 Case Study

We illustrate the effectiveness of IntentCoder through a case study presented in Fig. 5, In this case, IntentCoder correctly identifies the user's intent, which is to set a custom renderer for the table corner header (*e.g.*, `MultiLineTableHeaderRenderer`). In contrast, many other models (*e.g.*, DeepSeek-Coder, Qwen2.5-Coder, CodeLlama-7B, Starcoder2-7B, aixcoder-7B-v2), mistakenly think the next step should be related to the corner setup. Additionally, due to the lack of intent recognition's capability, many other models overly focus on the in-file context (*e.g.*, colored in blue), such as `FrozenColumnTable`, `setGridColor`, `getColumn`. At the same time, these models lack cross-file reasoning ability and incorrectly invoke classes from other files (*e.g.*, colored in purple), failing to understand that the appropriate multi-line header renderer should be used. By recognizing the overall functional requirements, IntentCoder accurately provides the correct code completion, avoiding the shortcomings of other models in completion intent recognition and cross-file reasoning.