From Completion to Editing: Unlocking Context-Aware Code Infilling via Search-and-Replace Instruction Tuning

Jiajun Zhang[1,3†] Zeyu Cui[2†‡] Jiaxi Yang[4] Lei Zhang[4] Yuheng Jing[3] Zeyao Ma[2]
Tianyi Bai[2] Binyuan Hui[2] Qiang Liu[3] Zilei Wang[1] Liang Wang[3‡] Junyang Lin[2]
[1]USTC [2]Alibaba Group [3]CASIA [4]SIAT
[†] Equal contribution. [‡] Corresponding authors.
zhangjiajun519@gmail.com
Project Page: Qwen3-Coder

Abstract

The dominant Fill-in-the-Middle (FIM) paradigm for code completion is constrained by its rigid inability to correct contextual errors and reliance on unaligned, insecure Base models. While Chat LLMs offer safety and Agentic workflows provide flexibility, they suffer from performance degradation and prohibitive latency, respectively. To resolve this dilemma, we propose Search-and-Replace Infilling (**SRI**), a framework that internalizes the agentic verification-and-editing mechanism into a unified, single-pass inference process. By structurally grounding edits via an explicit search phase, SRI harmonizes completion tasks with the instruction-following priors of Chat LLMs, extending the paradigm from static infilling to dynamic context-aware editing. We synthesize a high-quality dataset, **SRI-200K**, and fine-tune the **SRI-Coder** series. Extensive evaluations demonstrate that with minimal data (20k samples), SRI-Coder enables Chat models to surpass the completion performance of their Base counterparts. Crucially, unlike FIM-style tuning, SRI preserves general coding competencies and maintains inference latency comparable to standard FIM. We release our dataset and models, establishing SRI as a robust, secure, and efficient alignment recipe for next-generation interactive development.

1 Introduction

Large Language Models (LLMs) have revolutionized software development, with the Fill-in-the-Middle (FIM) paradigm serving as the industry standard for code completion. By conditioning on both prefix and suffix contexts, FIM enables models to generate missing code segments effectively (Bavarian et al., 2022). The FIM strategy has significantly advanced the code completion capabilities of numerous LLMs, including CodeX (Chen et al., 2021a), StarCoder (Lozhkov et al., 2024), DeepSeekCoder (Guo et al., 2024a), Qwen-Coder (Hui et al., 2024), and CodeLlama (Rozière et al., 2023). Leveraging these models, LLM-powered programming tools such as *GitHub Copilot*[1], *Augment*[2] and *Continue.dev*[3] have implemented powerful autocomplete features, which leverages context-aware suggestions and a convenient "*Tab-to-complete*" interface to boost developer productivity.

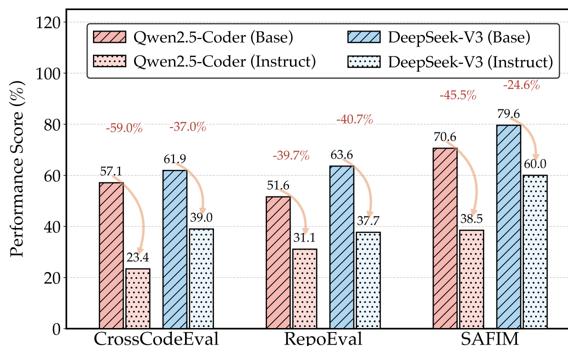

Figure 1: Performance disparity between Base models and their Instruction-tuned counterparts on standard code completion benchmarks. The results highlight a significant degradation in completion capabilities (up to 59.0%) when prompting safety-aligned Chat LLMs for FIM tasks, demonstrating that FIM skills do not naturally transfer to the conversational setting.

However, this paradigm exhibits two critical drawbacks: ❶ **The Optimal Context Assumption:** FIM operates on the unrealistic premise that the surrounding context is error-free. Consequently, when the context contains bugs, FIM is incapable of correcting them; it merely builds upon the flaws, inevitably resulting in broken code. ❷ **Inherent Security Risks:** FIM typically relies on unaligned base LLMs, which lack the safety alignments of chat LLMs. This exposes systems to severe vulnerabilities; notably, the SAL benchmark demonstrates that FIM-based tools like *GitHub Copilot*

[1]https://github.com/features/copilot
[2]https://docs.augmentcode.com
[3]https://www.continue.dev

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 7955–7984
July 2-7, 2026 ©2026 Association for Computational Linguistics

suffer from attack success rates exceeding 99% (Cheng et al., 2025). While directly prompting safety-aligned Chat LLMs to perform FIM tasks might seem like a solution, Figure 1 illustrates that this approach causes significant performance degradation, failing to effectively transfer completion capabilities to the instruction-tuned setting.

The conventional FIM paradigm is constrained by inherent rigidity and security vulnerabilities, while directly utilizing Chat LLMs for infilling often results in performance degradation. Furthermore, although multi-step agentic workflows offer enhanced capabilities, they incur latency overheads incompatible with real-time completion scenarios. To resolve this dilemma, we propose **S**earch-and-**R**eplace **I**nfilling (**SRI**), a framework that internalizes the agentic verification-and-editing mechanism into a unified, single-pass inference process. By structurally grounding edits through an explicit search phase, SRI not only harmonizes completion tasks with the instruction-following priors of Chat LLMs but also extends the paradigm from static infilling to dynamic context-aware editing. This end-to-end approach synthesizes the high performance and flexibility of agentic editing with the computational efficiency essential for interactive development.

Methodologically, we first substantiate the intrinsic advantages of SRI in mitigating security vulnerabilities and resolving contextual inconsistencies through controlled experiments (§3). Building on this foundation, we synthesize **SRI-200K**, a high-quality dataset derived from *The Stack v2* (Lozhkov et al., 2024), and construct the **SRI-Coder** series by fine-tuning the Qwen2.5-Coder family. We then conduct comprehensive evaluations across over 20 LLMs on both mainstream code completion and general coding benchmarks. Our empirical results yield four critical findings: ❶ SRI consistently and significantly outperforms various natural language FIM prompting strategies across Chat LLMs (§5.1); ❷ with minimal fine-tuning (20k samples), SRI-Coder enables Chat models to surpass the completion performance of their corresponding Base counterparts (§4.2.1); ❸ unlike FIM-style fine-tuning, which degrades general coding capabilities, SRI acts as a superior tuning format that enhances completion skills while preserving the model's broader competencies (§5.1); and ❹ crucially, SRI maintains inference latency comparable to standard FIM, validating its feasibility for real-time deployment despite the structured generation process (§5.2). Collectively, these findings establish SRI as a robust and efficient alternative for next-generation auto-completion tools, offering a promising direction for aligning large language models with interactive development workflows.

2 Preliminary

Fill-in-the-middle Code Completion. Fill-in-the-middle (FIM) is a pre-training strategy and inference format designed for base large language models (LLMs). It has been widely adopted in code-centric LLMs such as Qwen-Coder (Hui et al., 2024; Guo et al., 2024a; Lozhkov et al., 2024) and even some general-purpose models like DeepSeek-V3 (DeepSeek-AI and etc., 2024). FIM addresses a core limitation of decoder-only architectures—their inability to perform infilling tasks—through a data transformation technique applied during both training and inference. This allows the model to leverage both prefix and suffix contexts to generate an intermediate code segment.

During FIM pre-training, each document is split into three parts: `prefix`, `middle`, and `suffix`. These components are independently encoded and marked with special sentinel tokens `<PRE>`, `<MID>`, and `<SUF>`, respectively. The final tokenized sequence follows a specific order: `prefix` → `suffix` → `middle`, with their corresponding sentinel tokens preserved, as shown in Equation 1:

$$\texttt{<PRE>} \circ \texttt{prefix} \circ \texttt{<SUF>} \circ \texttt{suffix} \circ \texttt{<MID>} \circ \texttt{middle} \quad (1)$$

where ∘ denotes concatenation. During inference, given a prefix and suffix pair, we construct the input prompt by removing the middle segment and its content while maintaining the special tokens, as expressed in Equation 2:

$$\texttt{<PRE>} \circ \texttt{prefix} \circ \texttt{<SUF>} \circ \texttt{suffix} \circ \texttt{<MID>} \quad (2)$$

This structured prompt guides the model to synthesize a coherent middle segment by conditioning on both the prefix and suffix. By effectively resolving the infilling challenge for decoder-only LLMs, FIM has been instrumental in the advancement of LLM-powered autocomplete coding tools.

Code Completion in Chat LLMs While Base LLMs perform FIM via raw text continuation, Chat LLMs are constrained by post-training alignment formats (e.g., `ChatML` (OpenAI, 2022)). The mandatory application of chat templates (`apply_chat_template`) structurally conflicts

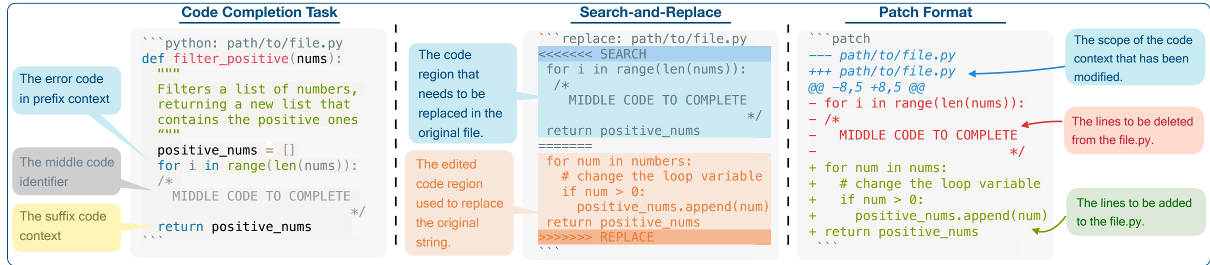

Figure 2: The workflow of our proposed Search-and-Replace Infilling (SRI) method. (1) The process begins with a code completion task containing a pre-existing bug (an incorrect loop variable) and an `/* MIDDLE CODE TO COMPLETE */` identifier marking the target location. (2) Our SRI model generates a search-and-replace block that simultaneously infills the missing function body and corrects the contextual bug. (3) This block can then be converted into a standard patch format for versatile application.

with the special tokens required for standard FIM inference, rendering direct application infeasible.

To bypass this limitation, practitioners typically employ natural language (NL) prompts to simulate FIM tasks. Common adaptations include *Standard FIM Completion* (Figure 11), *Dialogue-Based Reconstruction* (Figure 12), and *Template-Based Infilling* (Figure 13). However, the completion proficiency acquired during pre-training fails to effectively transfer to these conversational settings, resulting in significant performance degradation compared to Base models. A detailed theoretical analysis in Appendix E.2 further elucidates the inherent limitations of relying on unstructured NL instructions for precise code generation.

The Search-and-Replace Format The `diff` format, used to represent the differences between two files, is widely employed in applications such as version control (e.g., Git commits), code comparison tools, and coding agents. A common implementation is the Unified Diff Format, often referred to as a `patch`. A patch is a text file that declaratively specifies the changes required to transform a file from one state to another (e.g., "*delete these lines, add those lines*"). It precisely identifies changes based on line numbers and context. However, generating a syntactically correct patch is a significant challenge for LLMs, as they often struggle to produce accurate line numbers (Zhang et al., 2025). To address this limitation, Wei et al. (2025) proposed a Search-and-Replace format as a more robust alternative. This format, inspired by the GitHub merge conflict convention, uses explicit `SEARCH` and `REPLACE` blocks to delineate the code to be modified and its replacement. This structure effectively captures code edits by clearly defining the `before` and `after` states without relying on fragile line numbers. Its clear structure and high readability make it highly accessible to both humans and LLMs, leading to its widespread adoption in LLM-based software engineering tasks (Xia et al., 2024; Jimenez et al., 2024; Wang et al., 2024b).

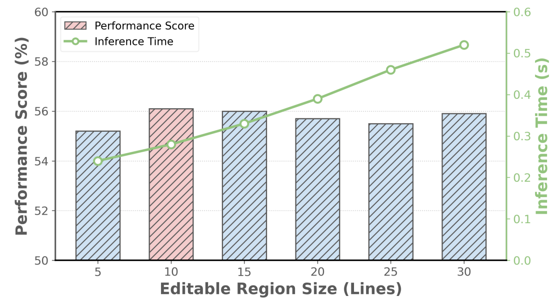

Figure 3: SRI editable region size sensitivity analysis.

3 From Static Completion to Context-Aware Editing

In this section, we introduce **Search-and-Replace Infilling (SRI)**, a paradigm shift that transitions code completion from static text continuation to intelligent micro-editing. We structure our discussion as follows: Section 3 details the grounding mechanism behind SRI, Section 3.2 demonstrates how SRI bridges the safety gap, and Section 3.3 showcases its robustness to contextual noise.

3.1 Methodology: Contextual Grounding

To address the limitations of static FIM infilling, we propose <u>S</u>earch-and-<u>R</u>eplace <u>I</u>nfilling (SRI). Unlike FIM, which generates the middle segment without structural validation, SRI reframes the task as **context-aware micro-editing**. The core innovation lies in its workflow, as illustrated in Figure 2. In the SRI framework, we designate the target location with a unique identifier, `/* MIDDLE CODE TO COMPLETE */`. Functionally analogous to the cursor in standard auto-completion tools, this marker explicitly denotes the

active position for generation. We then define an *editable region* comprising the identifier and 10 lines of surrounding context (Figure 3). Crucially, rather than immediately generating the new code, the model is prompted to first generate a `SEARCH` block that explicitly replicates the code segment intended for modification.

This `SEARCH` generation functions as a **grounding mechanism**, operating akin to a visual Chain-of-Thought. It compels the model to identify and validate the context before committing to any changes. Following this verification step, the model generates the `REPLACE` block. This structured output allows for direct application via IDE tools or conversion into standard patch formats (e.g., `git apply`). We select the search-and-replace format over standard diff patches for its **robustness**. While standard patches rely on fragile line numbers (which LLMs struggle to count accurately (Zhang et al., 2025)), the search-and-replace format anchors edits based on semantic content. This ensures reliable execution even when the model's internal line counting is imprecise.

3.2 Bridging the Security Alignment Gap

The traditional FIM paradigm forces the use of unaligned Base models, leaving systems inherently vulnerable to code injection. SRI addresses this critical flaw by serving as an **alignment bridge**, enabling code completion tasks to utilize instruction-tuned Chat models and effectively **inherit** their robust post-training safety features.

To validate this, we utilize the SAL benchmark (Cheng et al., 2025) to evaluate resistance against injection attacks. We compare the vulnerability of state-of-the-art Base models (using FIM) against their aligned Instruct counterparts (using SRI). The results for Level 1 (direct) and Level 2 (complex) attacks are presented in Table 1, with additional benchmark results detailed in Appendix §B. The data reveals a fundamental disparity: Base models are nearly defenseless, exhibiting attack success rates approaching 100%. In contrast, SRI-enabled models leverage the safety alignment of their Chat backbones to drastically reduce vulnerability. Notably, this architecture also allows for the integration of *Safe Prompts* (unlike FIM), which can further suppress successful attacks to near-zero levels.

3.3 Robustness to Contextual Noise

Real-world development environments are rarely perfect; they often contain contextual inconsistencies, such as syntax errors, legacy patterns, or temporary placeholders. Traditional FIM operates under a rigid assumption of frozen context, merely appending code without addressing existing flaws. In contrast, SRI exhibits **intent-aware robustness**, enabling the model to dynamically rectify the surrounding context to match the user's objective. To measure this capability, we construct **CrossCodeEval-Flex**. This benchmark adapts the standard CrossCodeEval dataset (Ding et al., 2023b) by simulating common developer error patterns. Specifically, we introduce controlled syntactical or logical noise into the 5-line window surrounding the missing middle code. This setup creates a challenging scenario where the model must perform simultaneous completion and correction. The results in Table 2 highlight a critical disparity across diverse model families. Standard FIM models, including both **Qwen2.5-Coder-32B** and the state-of-the-art **DeepSeek-V3-Base**, fail completely and yield near 0.0 EM scores as they are architecturally constrained to preserve the noisy context. In contrast, the Chat LLMs utilizing SRI successfully identify and resolve the inconsistencies. Specifically, **SRI-Coder-32B**, the model we fine-tuned for this paradigm, achieves the most significant improvements. This confirms that SRI transcends the limitations of static completion, offering a resilient solution for complex editing scenarios regardless of the underlying model architecture.

Model	Attack Success Rate (%)	
	Level 1	Level 2
Commercial Baseline		
GitHub Copilot	98.8	99.4
Qwen2.5-Coder Series		
Qwen-Base (FIM)	100.0	97.5
Qwen-Instruct (SRI)	**13.8**	**20.0**
DeepSeek-V3 Series		
DeepSeek-Base (FIM)	98.2	95.4
DeepSeek-Instruct (SRI)	**5.5**	**7.1**

Table 1: Security evaluation on the SAL Benchmark. The table compares commercial/base FIM models against aligned Instruct models. Even leading commercial tools like Copilot are vulnerable, whereas SRI effectively bridges the alignment gap.

4 Experiments

4.1 Experimental Setup

Training Dataset To ensure robust evaluation, our training data underwent strict decontamination to prevent repository-level overlap with bench-

Model	CrossCodeEval-Flex	
	EM	ES
Qwen2.5-Coder Series		
Qwen-Base (FIM)	0.0	35.6
Qwen-Instruct (SRI)	14.0	53.8
SRI-Coder	**33.0**	**71.5**
DeepSeek-V3 Series		
DeepSeek-Base (FIM)	0.0	38.2
DeepSeek-Chat (SRI)	24.5	67.4

Table 2: Performance on CrossCodeEval-Flex. We evaluate both the Qwen2.5-Coder-32B series and the DeepSeek-V3 series. Standard FIM fails completely in noisy environments (0.0 EM), whereas SRI successfully acts as a corrective editor. Note that SRI-Coder refers to the model fine-tuned on our synthesized dataset.

marks. The dataset integrates two distinct components: the novel **SRI-200K** and a general instruction set. For the SRI component, we sourced code from *The Stack v2* (Lozhkov et al., 2024) and employed tree-sitter to extract fundamental logic blocks (e.g., functions, loops) as infilling targets. These were formulated into search-and-replace queries with a maximum context length of 32k tokens. While we constructed 200k samples in total, we utilized a 20k subset for fine-tuning and created a markdown-formatted variant for ablation studies. Complementing this, we sampled 60k instruction pairs from *Glaive-Code-Assistant* (Glaive AI, 2023), meticulously filtering out completion-related tasks to maintain general instruction-following capabilities (Appendix §F).

Benchmarks To rigorously evaluate code completion, we employ five mainstream benchmarks originally designed for FIM tasks. These include three similarity-based metrics: CrossCodeEval (Ding et al., 2023b), CrossCodeLongEval (Wu et al., 2024), and RepoEval (Zhang et al., 2023), and two execution-based benchmarks: SAFIM (Gong et al., 2024) and ExecRepoBench (Yang et al., 2024a). Spanning diverse sources like GitHub and Codeforces, this suite comprehensively covers both repository-level and in-file scenarios.

Training Details We fine-tuned Qwen2.5-Coder Base models using Megatron-LM (Shoeybi et al., 2020) on 16 NVIDIA A100 GPUs. Training utilized a 32k context length, global batch size of 256, and BFloat16 precision. The learning rate followed a linear decay from 5×10^{-5} to 5×10^{-6} over 853 steps. Crucially, to strictly isolate the impact of our method and prevent contamination from prior alignment, we trained starting from base checkpoints using our 80k mixed dataset (20k SRI, 60k general instructions, 100 safety). Other architectural configurations align with the standard Qwen2.5-Coder implementation (Hui et al., 2024).

Evaluation Details We evaluated a diverse array of models, including the GPT (OpenAI, 2023), Claude (Anthropic, 2023a), Gemini (Team, 2024), DeepSeek (V3/R1) (DeepSeek-AI and etc., 2024; DeepSeek-AI et al., 2025), and Qwen (2.5/3) (team and etc., 2025; Yang et al., 2025) series. Open-source models were deployed locally via vLLM (using 2x A100s for Qwen2.5-Coder), while proprietary models were accessed via official APIs. Inference utilized greedy decoding with a 256-token limit. We employed three distinct prompting strategies: standard token-based FIM for Base models, and both Natural Language prompting and our proposed SRI format for Chat models (see Figure 14, 11, 12, 13). Context lengths were standardized to 32k for ExecRepoBench, 8k for similarity benchmarks, and full context for SAFIM. To ensure strict fairness, we maintained identical prefix/suffix contexts across all methods. Furthermore, we inserted a `/* MIDDLE CODE TO COMPLETE */` identifier in SRI prompts to explicitly mark the generation target, equalizing the positional information provided implicitly by FIM and NL prompts. Further details are in Appendix §C.

4.2 Results and Analysis

4.2.1 Main Results

We conduct a comprehensive evaluation across five mainstream benchmarks using three distinct inference paradigms: ① Special token-based FIM (**FIM**); ② Natural language-based completion (**Chat-FIM**), for which we report the **maximum score** achieved across the three prompting variants (Standard, Dialogue, Template); and ③ our proposed Search-and-Replace Infilling (**SRI**). Tables 6 and 7 summarize the performance across 20+ leading models (full details in Appendix §B). Our analysis reveals several key findings: ❶ **SRI Outperforms Chat-FIM**. SRI consistently yields significant gains over the best-performing Chat-FIM baselines. However, in a zero-shot setting, even the top Chat models still lag behind state-of-the-art (SOTA) Base models like DeepSeek-V3. ❷ **Effectiveness of SRI Fine-Tuning**. Our fine-tuned **SRI-Coder-32B** surpasses its Base counterpart (Qwen2.5-Coder-32B) on all five benchmarks.

Model	CrossCodeEval				RepoEval				CrossCodeLongEval			
	EM		*ES*		*EM*		*ES*		*EM*		*ES*	
	FIM	**SRI**	**FIM**	**SRI**	**FIM**	**SRI**	**FIM**	**SRI**	**FIM**	**SRI**	**FIM**	**SRI**
Base Models												
Qwen2.5-Coder-32B	57.1	-	86.8	-	51.6	-	78.5	-	36.8	-	66.4	-
DeepSeek-V3-Base	61.9	-	88.5	-	63.6	-	86.2	-	50.5	-	77.9	-
Non-reasoning Chat Models												
DeepSeek-V3	39.0	56.1 ↑17.1	70.3	86.2 ↑15.9	37.7	54.3 ↑16.6	60.4	78.9 ↑18.5	25.4	34.8 ↑9.4	53.5	67.4 ↑13.9
DeepSeek-V3-0324	44.5	56.0 ↑11.5	80.0	85.7 ↑5.7	45.8	54.5 ↑8.7	68.2	79.3 ↑11.1	30.9	34.8 ↑3.9	60.2	68.2 ↑8.0
Claude-3.5-Haiku	23.9	44.5 ↑20.6	73.2	75.2 ↑2.0	31.9	44.4 ↑12.5	58.2	67.8 ↑9.6	17.7	27.1 ↑9.4	49.8	56.2 ↑6.4
Claude-3.5-Sonnet	47.5	56.0 ↑8.5	82.3	85.6 ↑3.3	48.0	54.4 ↑6.4	72.1	80.2 ↑8.1	32.4	36.3 ↑3.9	63.2	69.4 ↑6.2
Claude-3.7-Sonnet	48.3	56.7 ↑8.4	81.6	85.7 ↑4.1	49.3	55.0 ↑5.7	73.6	80.2 ↑6.6	32.5	38.2 ↑5.7	64.0	70.5 ↑6.5
Claude-4-Sonnet	57.5	60.8 ↑3.3	87.3	87.2 ↓0.1	55.7	58.9 ↑3.2	78.7	82.0 ↑3.3	38.1	38.9 ↑0.7	68.8	69.5 ↑0.7
GPT-4o-1120	34.2	44.1 ↑9.9	72.5	80.9 ↑8.4	25.7	45.0 ↑19.3	52.1	72.6 ↑20.5	21.7	23.6 ↑1.9	51.6	56.4 ↑4.8
GPT-4o-0806	40.3	41.7 ↑1.4	77.3	79.5 ↑2.2	37.2	46.8 ↑9.6	61.5	73.7 ↑12.2	20.6	26.4 ↑5.8	50.7	58.2 ↑7.5
GPT-4o-mini	9.0	27.7 ↑18.7	59.6	68.3 ↑8.7	19.6	27.7 ↑8.1	48.0	54.6 ↑6.6	12.1	11.1 ↓1.0	41.3	39.6 ↓1.7
GPT-4.1	46.1	45.8 ↓0.3	81.0	79.8 ↓1.2	47.0	46.9 ↓0.1	71.7	73.2 ↑1.5	28.5	30.8 ↑2.3	61.2	62.2 ↑1.0
Qwen3-Coder-480B-A35B	56.1	**62.9** ↑6.9	85.8	**89.5** ↑3.7	51.1	**65.1** ↑14.0	73.2	**85.4** ↑12.2	36.2	**45.9** ↑9.7	65.3	**74.7** ↑9.4
Qwen2.5-Coder-32B-Instruct	23.4	44.4 ↑21.0	69.9	76.8 ↑6.9	31.1	44.8 ↑13.7	56.1	69.7 ↑13.6	20.4	25.9 ↑5.5	50.3	55.8 ↑5.5
SRI-Coder-32B	11.3	57.6 ↑46.3	37.9	87.3 ↑49.3	17.3	54.4 ↑37.1	37.3	78.9 ↑41.5	14.6	37.1 ↑22.5	33.2	67.9 ↑34.7
Reasoning Chat Models												
DeepSeek-R1-0528	44.6	51.1 ↑6.5	82.3	84.0 ↑1.7	48.1	49.2 ↑1.1	75.1	76.0 ↑0.9	29.2	29.7 ↑0.5	62.2	62.6 ↑0.4
Claude-3.7-Sonnet-think	49.4	**54.6** ↑5.2	82.5	**84.5** ↑2.0	9.1	53.0 ↑43.9	27.6	78.0 ↑50.4	30.6	34.9 ↑4.3	60.7	66.9 ↑6.2
Grok-3	46.4	54.4 ↑8.0	82.1	84.1 ↑2.0	50.1	**53.6** ↑3.5	75.8	**78.7** ↑2.9	32.4	**35.2** ↑2.8	65.2	**68.0** ↑2.8
o1-2024-12-17	26.9	42.7 ↑15.8	75.4	78.4 ↑3.0	44.5	48.3 ↑3.8	68.4	73.6 ↑5.2	26.6	24.4 ↓2.2	56.6	57.0 ↑0.4
o3-mini	23.7	27.6 ↑3.9	39.1	69.9 ↑30.8	26.4	39.1 ↑12.7	37.1	61.4 ↑24.3	9.7	20.9 ↑11.2	21.1	48.9 ↑27.8
Qwen3-30B-A3B	13.7	36.1 ↑22.4	59.7	71.5 ↑11.8	20.0	33.6 ↑13.6	44.3	61.2 ↑16.9	12.9	17.6 ↑4.7	36.7	45.1 ↑8.4
Qwen3-235B-A22B	14.6	42.7 ↑28.2	35.6	76.8 ↑41.3	19.1	42.7 ↑23.6	30.6	68.9 ↑38.3	15.0	23.2 ↑8.2	32.5	53.5 ↑21.0

Table 3: Performance comparison on similarity-based benchmarks (CrossCodeEval, RepoEval, and CrossCode-LongEval). For base models, **FIM** denotes special token based fill-in-the-middle code completion, for chat models, **FIM** denotes natural language-based code completion prompting, while **SRI** represents our method. **Bold** and underlined values indicate the best and second-best performance respectively within the model type. ↑n and ↓n indicate performance improvements and degradations of SRI compared to FIM.

Crucially, it substantially outperforms the original Instruct version across all metrics, rivaling larger models like DeepSeek-V3 and closed-source commercial APIs. This validates the high efficacy of our synthesized training data. ❸ **Model Specialization**. We observe a trade-off in model specialization: reasoning-heavy models excel at short-context algorithmic tasks but often struggle with long-context, repository-level benchmarks.

5 Discussion

5.1 Ablation Studies

To isolate the efficacy of the SRI format, we conducted a rigorous comparative analysis against NL-FIM based fine-tuning strategies. We trained our primary model, **SRI-Coder-32B**, and compared it against three variants fine-tuned on the *identical* 20k dataset but reformatted into distinct NL-FIM paradigms: ❶ **Standard FIM Completion**: Direct instructions explicitly asking the model to complete the missing code; ❷ **Dialogue-Based FIM Reconstruction**: Multi-turn interactions simulating a user seeking code restoration; ❸ **Template-Based FIM Infilling**: Fixed prompt templates guiding the model to fill in specific slots.

We evaluated these models on our five code completion benchmarks alongside a suite of general coding benchmarks (MBPP, HumanEval, BigCodeBench, and LiveCodeBench). The results (Figure 1) reveal two critical findings. First, **SRI-Coder-32B** achieves a superior average completion score of 56.1, consistently outperforming all NL-FIM variants. Second, NL-based FIM strategies result in a measurable performance regression on general coding benchmarks, whereas **SRI-Coder** effectively preserves the model's broader competencies. We hypothesize that this degradation arises because NL-FIM conditions the model to generate fragmented code segments, a pattern detrimental to the structural coherence required for general programming tasks. Consequently, we attribute SRI's superiority to its alignment with the extensive diff-formatted data inherent in pre-training corpora,

Model	ExecRepoBench		SAFIM									
	Pass@1		*Block*		*Block V2*		*API*		*Control*		*Average*	
	FIM	**SRI**	**FIM**	**SRI**	**FIM**	**SRI**	**FIM**	**SRI**	**FIM**	**SRI**	**FIM**	**SRI**
Base Models												
Qwen2.5-Coder-32B	25.7	-	62.9	-	65.2	-	75.5	-	76.4	-	70.0	-
DeepSeek-V3-Base	36.5	-	76.3	-	77.9	-	79.4	-	84.8	-	79.6	-
Non-reasoning Chat Models												
DeepSeek-V3	35.6	61.8 ↑26.2	60.5	**70.1** ↑9.6	59.7	70.6 ↑10.9	55.8	73.6 ↑17.8	64.1	80.6 ↑16.5	60.0	73.7 ↑13.7
DeepSeek-v3-0324	36.9	65.5 ↑28.6	53.3	69.8 ↑16.5	56.5	71.1 ↑14.6	68.1	74.5 ↑6.4	65.3	**81.8** ↑16.5	60.8	74.3 ↑13.5
Claude-3.5-Haiku	35.6	57.7 ↑22.1	60.1	66.3 ↑6.2	59.7	68.0 ↑8.3	65.2	71.6 ↑6.4	68.5	77.4 ↑8.9	63.4	70.8 ↑7.4
Claude-3.5-Sonnet	41.8	66.2 ↑24.4	60.6	65.9 ↑5.3	60.2	68.0 ↑7.8	61.3	64.2 ↑2.9	68.9	75.2 ↑6.3	62.7	68.3 ↑5.6
Claude-3.7-Sonnet	42.0	62.8 ↑20.8	61.6	67.8 ↑6.2	60.9	70.5 ↑9.6	64.2	66.1 ↑1.9	66.1	79.3 ↑13.2	63.2	70.9 ↑7.7
Claude-4-Sonnet	61.2	67.1 ↑5.9	67.3	69.8 ↑2.4	69.2	**72.9** ↑3.7	72.9	74.2 ↑1.3	78.6	80.6 ↑2.0	72.0	**74.4** ↑2.4
GPT-4o-1120	36.2	55.9 ↑19.7	59.5	62.4 ↑2.9	55.9	62.3 ↑6.4	65.2	68.1 ↑2.9	58.6	75.4 ↑16.8	59.8	67.0 ↑7.2
GPT-4o-0806	39.4	58.4 ↑19.0	47.9	59.7 ↑11.8	46.6	57.0 ↑10.4	64.2	69.4 ↑5.2	54.9	67.3 ↑12.4	53.4	63.4 ↑10.0
GPT-4o-mini	28.0	40.8 ↑12.8	25.2	32.4 ↑7.2	26.8	32.4 ↑5.6	23.9	44.8 ↑20.9	9.5	43.4 ↑33.9	21.3	38.2 ↑16.9
GPT-4.1	41.4	59.6 ↑18.2	51.2	66.6 ↑15.4	53.1	63.5 ↑10.4	68.1	71.3 ↑3.2	59.8	68.6 ↑8.8	58.0	67.5 ↑9.5
Qwen3-Coder-480B-A22B	52.1	**79.1** ↑27.0	63.4	66.3 ↑2.9	65.5	69.6 ↑4.1	74.8	**78.1** ↑3.2	74.6	81.2 ↑6.6	69.6	73.8 ↑4.2
Qwen2.5-Coder-32B-Instruct	43.6	46.0 ↑2.4	36.8	44.9 ↑8.1	37.8	48.6 ↑10.8	48.4	68.4 ↑20.0	31.1	55.8 ↑24.7	38.5	54.4 ↑15.9
SRI-Coder-32B	24.5	61.6 ↑37.1	10.3	60.5 ↑50.1	15.3	64.5 ↑49.2	5.3	74.9 ↑69.6	8.0	75.6 ↑67.6	9.8	69.9 ↑60.1
Reasoning Chat Models												
DeepSeek-R1-0528	42.1	60.1 ↑18.0	62.3	69.8 ↑7.5	60.5	**72.3** ↑11.8	57.1	**75.9** ↑18.8	66.2	**79.8** ↑13.6	61.5	**74.5** ↑13.0
Claude-3.7-Sonnet-think	44.2	61.3 ↑17.1	64.3	68.3 ↑4.0	63.6	70.7 ↑7.1	58.4	66.8 ↑8.4	72.9	79.2 ↑6.3	64.8	71.3 ↑6.5
Grok-3	39.1	**63.4** ↑24.3	58.7	63.3 ↑4.6	59.1	66.6 ↑7.5	61.9	72.3 ↑10.4	65.5	73.8 ↑8.3	61.3	69.0 ↑7.7
o1-2024-12-17	41.1	51.5 ↑10.4	62.6	68.7 ↑6.1	65.9	71.1 ↑5.2	67.1	72.2 ↑5.1	65.9	74.4 ↑8.5	65.4	71.6 ↑6.2
o3-mini	43.0	57.0 ↑14.0	32.1	**76.4** ↑44.3	28.0	71.6 ↑43.6	44.8	57.7 ↑12.9	38.2	73.1 ↑34.9	35.8	69.7 ↑34.1
Qwen3-30B-A3B	33.0	48.5 ↑15.5	29.9	42.6 ↑12.7	34.7	48.0 ↑13.3	50.0	64.8 ↑14.8	34.1	58.3 ↑24.2	37.2	53.4 ↑16.2
Qwen3-235B-A22B	35.6	54.7 ↑19.1	33.3	47.5 ↑12.6	36.1	48.7 ↑12.6	35.2	63.5 ↑28.4	52.6	58.1 ↑5.5	39.3	54.4 ↑15.2

Table 4: Performance comparison on unit tests-based benchmarks (ExecRepoBench and SAFIM). In SAFIM, all metrics are reported as Pass@1 except API which uses Exact Match (EM).

which minimizes the distributional shift observed in NL-FIM strategies (analysis in Appendix §H).

5.2 Scaling Studies

To investigate the effect of model scale on the performance of our SRI fine-tuning, we conducted a series of scaling experiments. Using the same mixed SFT dataset (20K SRI and 60K Glaive-Code samples), we fine-tuned the entire Qwen2.5-Coder model series, spanning parameters from 0.5B, 1.5B, 3B, 7B, 14B, to 32B. We then evaluated two sets of models on our five code completion benchmarks: (Appendix §B, Table 8,9) ❶ The Qwen2.5-Coder base models, using the standard FIM inference method. ❷ Our fine-tuned SRI-Coder models, using our proposed SRI inference method. The average scores and inference times across all five benchmarks are presented in Figure 5. Our analysis yields two primary conclusions: ❶ **Scaling Advantage of SRI**: The performance advantage of SRI over FIM grows as the model size increases. Larger models appear to better master the structured search-and-replace task, whereas smaller models struggle to a greater extent. ❷ **Comparable Inference Speed**: For models of the same size, the inference speed of SRI is nearly identical to that of FIM.

6 Related Works

6.1 Large Language Models for Coding

LLMs have demonstrated unprecedented advances in coding capabilities in recent years. Leading proprietary LLMs, such as GPT (OpenAI, 2023) and Claude (Anthropic, 2023b), exhibit exceptional performance across diverse programming tasks. Specialized code-centric models (Chen et al., 2021a; Scao et al., 2022; Li et al., 2022; Allal et al., 2023; Fried et al., 2022; Jiang et al., 2024; Nijkamp et al., 2023; Wei et al., 2023a; Zhao et al., 2024; Lozhkov et al., 2024), including CodeLlama (Rozière et al., 2023), DeepSeek-Coder (Guo et al., 2024a), OpenCoder (Huang et al., 2024), and Qwen-Coder (Hui et al., 2024), have been developed to excel in specific tasks such as code debugging (Huq et al., 2022), translation (Jiao et al., 2023), and completion (Bavarian et al., 2022). These models employ domain-specific architectures and are trained on extensive code corpora, comprising billions of snip-

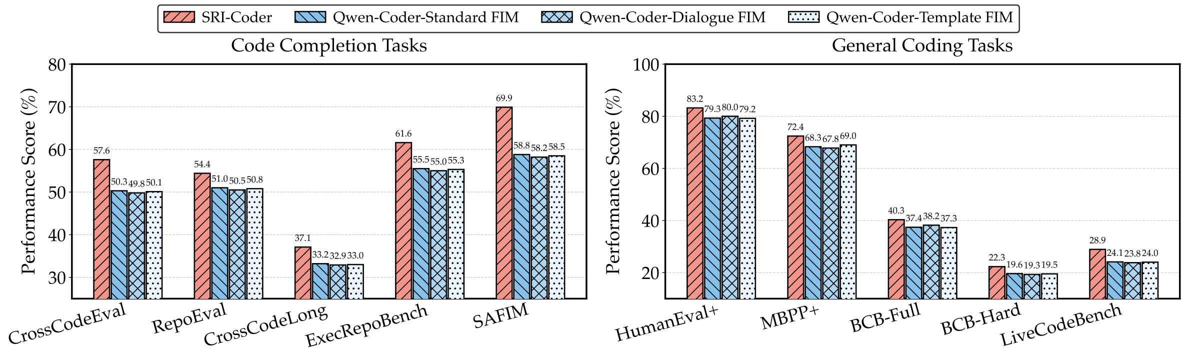

Figure 4: Ablation study comparing **SRI-Coder-32B** against three natural language-based FIM (NL-FIM) fine-tuning variants. **(Left)** On code completion benchmarks, SRI significantly outperforms the NL-FIM strategies. **(Right)** On general coding benchmarks, SRI preserves the model's broader competencies, whereas NL-FIM tuning induces measurable performance regression.

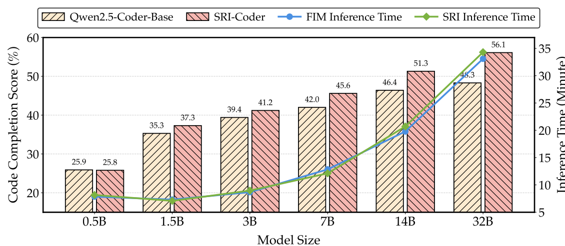

Figure 5: Scaling laws for FIM vs. SRI. Performance (left, %) and average inference time (right, minutes) across model sizes on code completion benchmarks.

pets, to optimize programming-related capabilities.

6.2 Code Completion and Coding Tools

In the context of code completion, the pioneering work of Bavarian et al. (2022) revolutionized the field by introducing the fill-in-the-middle (FIM) pre-training strategy. This approach implements a data transformation by relocating text spans from the middle of a document to its end. Models such as CodeX (Chen et al., 2021a), StarCoder (Lozhkov et al., 2024), DeepSeek-Coder (Guo et al., 2024a), Qwen-Coder (Hui et al., 2024; Cao et al., 2026; Team, 2025), and CodeLlama (Rozière et al., 2023) have integrated the FIM strategy into their training. Following this breakthrough, subsequent research has continuously advanced code completion capabilities (Chen et al., 2021a; Allal et al., 2023; Nijkamp et al., 2023; Jiang et al., 2024; Chen et al., 2024; Team) and enhanced training methodologies (Ding et al., 2024; Rozière et al., 2023). Code completion tasks require models to generate missing code segments by leveraging both left and right contexts, enabling the implementation of "*auto-complete*" features in LLM-powered coding tools.

Prominent LLM-powered programming tools such as *GitHub Copilot*, *Augment*, and *Continue.dev* have adopted FIM as their primary code completion solution, powering convenient "Tab-to-complete" functionality for developers. However, the recent rise of agentic coding has ushered in a new wave of agent-based tools, including *Curosr*, *Cline*, *Windsurf*, *Claude Code*, and *Gemini-Cli*. These tools can operate at the repository level, plan implementation strategies, and perform multi-file edits. Those capabilities are far exceed the scope of simple autocomplete. This evolution necessitates a more powerful and flexible code completion paradigm than FIM, one capable of supporting more generalized and intelligent programming assistance. A comprehensive review of additional related works is provided in Appendix A.

7 Conclusion and Future Directions

In this work, we identified the inherent limitations of FIM—rigidity, security vulnerabilities, and incompatibility with Chat models—and proposed **Search-and-Replace Infilling (SRI)** as a superior alternative. Through the release of our synthesized dataset and the **SRI-Coder** series, we demonstrate that minimal fine-tuning allows Chat models to surpass the completion capabilities of Base models. These findings challenge the dominance of FIM, suggesting that SRI provides a more secure and flexible foundation for next-generation, agentic coding tools. Looking forward, we aim to validate our approach by integrating SRI-Coder into functional IDEs to gather real-world developer feedback. Additionally, observing that smaller models (e.g., 0.5B) struggle with the structural complexity of the SRI task compared to their larger counterparts, we plan to explore techniques such as knowledge distillation to effectively transfer SRI

capabilities to these compact models.

8 Limitations

Despite the promising results, our work has two primary limitations. First, our current evaluation relies exclusively on offline benchmarks. While standardized metrics provide valuable insights, they cannot fully capture the dynamic and interactive nature of real-world development. We have not yet incorporated qualitative feedback from actual users or gathered telemetry data from functional IDE deployments. Second, we observe a performance disparity across model scales. While our SRI fine-tuning yields significant gains for larger models (e.g., 14B and 32B), the improvements are less pronounced for smaller architectures (e.g., 0.5B). This suggests that the structural complexity of the SRI task may require a certain threshold of model capacity to be fully effective.

9 Acknowledgments

This work is supported by National Key Research and Development Program (2023YFC3305203), National Natural Science Foundation of China (92570204, 62576339)

References

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, and 1 others. 2023. SantaCoder: Don't reach for the stars! *arXiv preprint arXiv:2301.03988*.

Anthropic. 2023a. Claude 2. Technical report, Anthropic.

Anthropic. 2023b. Introducing Claude.

Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. 2022. Efficient training of language models to fill in the middle. *arXiv preprint arXiv:2207.14255*.

Ruisheng Cao, Mouxiang Chen, Jiawei Chen, Zeyu Cui, Yunlong Feng, Binyuan Hui, Yuheng Jing, Kaixin Li, Mingze Li, Junyang Lin, and 1 others. 2026. Qwen3-coder-next technical report. *arXiv preprint arXiv:2603.00729*.

Saikat Chakraborty, Yangruibo Ding, Miltiadis Allamanis, and Baishakhi Ray. 2022. Codit: Code editing with tree-based neural models. *IEEE Transactions on Software Engineering*, 48(4):1385–1399.

Ge Chang, Jinbo Su, Jiacheng Liu, Pengfei Yang, Yuhao Shang, Huiwen Zheng, Hongli Ma, Yan Liang, Yuanchun Li, and Yunxin Liu. 2025. Grail: Learning to interact with large knowledge graphs for retrieval augmented reasoning. *arXiv preprint arXiv:2508.05498*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021a. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, abs/2107.03374.

Mouxiang Chen, Hao Tian, Zhongxin Liu, Xiaoxue Ren, and Jianling Sun. 2024. Jumpcoder: Go beyond autoregressive coder via online modification. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 11500–11520.

Mouxiang Chen, Lei Zhang, Yunlong Feng, Xuwu Wang, Wenting Zhao, Ruisheng Cao, Jiaxi Yang, Jiawei Chen, Mingze Li, Zeyao Ma, and 1 others. 2026. Swe-universe: Scale real-world verifiable environments to millions. *arXiv preprint arXiv:2602.02361*.

Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noel Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2021b. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, page 1–1.

Wen Cheng, Ke Sun, Xinyu Zhang, and Wei Wang. 2025. Security attacks on llm-based code completion tools. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 23669–23677.

DeepSeek-AI and etc. 2024. Deepseek-v3 technical report. *Preprint*, arXiv:2412.19437.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, and 181 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *Preprint*, arXiv:2501.12948.

Yue Deng, Wenxuan Zhang, Sinno Jialin Pan, and Lidong Bing. 2023. Multilingual jailbreak challenges in large language models. *arXiv preprint arXiv:2310.06474*.

Peng Ding, Jun Kuang, Dan Ma, Xuezhi Cao, Yunsen Xian, Jiajun Chen, and Shujian Huang. 2023a. A wolf in sheep's clothing: Generalized nested jailbreak prompts can fool large language models easily. *arXiv preprint arXiv:2311.08268*.

Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2023b. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Yifeng Ding, Hantian Ding, Shiqi Wang, Qing Sun, Varun Kumar, and Zijian Wang. 2024. Horizon-length prediction: Advancing fill-in-the-middle capabilities for code generation with lookahead planning. *Preprint*, arXiv:2410.03103.

Chengyu Fang, Heng Guo, Zheng Jiang, Chunming He, Xiu Li, and Minfeng Xu. 2026. Photon: Speedup volume understanding with efficient multimodal large language models. *arXiv preprint arXiv:2603.25155*.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida I. 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*, abs/2204.05999.

Glaive AI. 2023. glaive-code-assistant. `https://huggingface.co/datasets/glaiveai/glaive-code-assistant`.

Linyuan Gong, Sida Wang, Mostafa Elhoushi, and Alvin Cheung. 2024. Evaluation of llms on syntax-aware code fill-in-the-middle tasks. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, and 1 others. 2024a. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Xingang Guo, Fangxu Yu, Huan Zhang, Lianhui Qin, and Bin Hu. 2024b. Cold-attack: Jailbreaking llms with stealthiness and controllability. *arXiv preprint arXiv:2402.08679*.

Priyanshu Gupta, Avishree Khare, Yasharth Bajpai, Saikat Chakraborty, Sumit Gulwani, Aditya Kanade, Arjun Radhakrishna, Gustavo Soares, and Ashish Tiwari. 2023. Grace: Generation using associated code edits. *Preprint*, arXiv:2305.14129.

Divij Handa, Advait Chirmule, Bimal G Gajera, and Chitta Baral. 2024. Jailbreaking proprietary large language models using word substitution cipher. *CoRR*.

Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, and 1 others. 2023. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*.

Siming Huang, Tianhao Cheng, Jason Klein Liu, Jiaran Hao, Liuyihan Song, Yang Xu, J Yang, JH Liu, Chenchen Zhang, Linzheng Chai, and 1 others. 2024. Opencoder: The open cookbook for top-tier code large language models. *arXiv preprint arXiv:2411.04905*.

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

Tingfeng Hui, Pengyu Zhu, Bowen Ping, Ling Tang, Guanting Dong, Yaqi Zhang, and Sen Su. 2025. Decif: Improving instruction-following through meta-decomposition. *arXiv preprint arXiv:2505.13990*.

Faria Huq, Masum Hasan, Md Mahim Anjum Haque, Sazan Mahbub, Anindya Iqbal, and Toufique Ahmed. 2022. Review4repair: Code review aided automatic program repairing. *Information and Software Technology*, 143:106765.

Siyuan Jiang, Jia Li, He Zong, Huanyu Liu, Hao Zhu, Shukai Hu, Erlu Li, Jiazheng Ding, Yu Han, Wei Ning, Gen Wang, Yihong Dong, Kechi Zhang, and Ge Li. 2024. aixcoder-7b: A lightweight and effective large language model for code completion. *CoRR*, abs/2410.13187.

Zheng Jiang, Heng Guo, Chengyu Fang, Changchen Xiao, Xinyang Hu, Lifeng Sun, and Minfeng Xu. 2026. Medvr: Annotation-free medical visual reasoning via agentic reinforcement learning. In *The Fourteenth International Conference on Learning Representations*.

Mingsheng Jiao, Tingrui Yu, Xuan Li, Guanjie Qiu, Xiaodong Gu, and Beijun Shen. 2023. On the evaluation of neural code translation: Taxonomy and benchmark. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1529–1541. IEEE.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues? *Preprint*, arXiv:2310.06770.

Daniel Kang, Xuechen Li, Ion Stoica, Carlos Guestrin, Matei Zaharia, and Tatsunori Hashimoto. 2024. Exploiting programmatic behavior of llms: Dual-use through standard security attacks. In *2024 IEEE Security and Privacy Workshops (SPW)*, pages 132–143. IEEE.

Taewoo Kim and Joo-Haeng Lee. 2020. C-3po: Cyclic-three-phase optimization for human-robot motion retargeting based on reinforcement learning. *Preprint*, arXiv:1909.11303.

Jia Li, Ge Li, Zhuo Li, Zhi Jin, Xing Hu, Kechi Zhang, and Zhiyi Fu. 2023. <scp>codeeditor</scp> : Learning to edit source code with pre-trained models. *ACM Transactions on Software Engineering and Methodology*, 32(6):1–22.

Weichen Li, Albert Jan, Baishakhi Ray, Junfeng Yang, Chengzhi Mao, and Kexin Pei. 2025. Editlord: Learning code transformation rules for code editing. *Preprint*, arXiv:2504.15284.

Yujia Li, David H. Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, and 7 others. 2022. Competition-level code generation with AlphaCode. *arXiv preprint arXiv:2203.07814*, abs/2203.07814.

Chenyan Liu, Yufan Cai, Yun Lin, Yuhuan Huang, Yunrui Pei, Bo Jiang, Ping Yang, Jin Song Dong, and Hong Mei. 2024. Coedpilot: Recommending code edits with learned prior edit relevance, project-wise awareness, and interactive nature. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '24, page 466–478. ACM.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023a. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *arXiv preprint arXiv:2305.01210*, abs/2305.01210.

Xiaogeng Liu, Nan Xu, Muhao Chen, and Chaowei Xiao. 2023b. Autodan: Generating stealthy jailbreak prompts on aligned large language models. *arXiv preprint arXiv:2310.04451*.

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*.

Sijia Luo, Xiaokang Zhang, Yuxuan Hu, Bohan Zhang, Ke Wang, Jinbo Su, Mengshu Sun, Lei Liang, and Jing Zhang. 2026. Sparse-rl: Breaking the memory wall in llm reinforcement learning via stable sparse rollouts. *arXiv preprint arXiv:2601.10079*.

Xingjun Ma, Yifeng Gao, Yixu Wang, Ruofan Wang, Xin Wang, Ye Sun, Yifan Ding, Hengyuan Xu, Yunhao Chen, Yunhan Zhao, and 1 others. 2025. Safety at scale: A comprehensive survey of large model safety. *arXiv preprint arXiv:2502.05206*.

Zeyao Ma, Jing Zhang, Xiaokang Zhang, Jiaxi Yang, Zongmeng Zhang, Jiajun Zhang, Yuheng Jing, Lei Zhang, Hao Zheng, Wenting Zhao, and 1 others. 2026. Scaling agentic verifier for competitive coding. *arXiv preprint arXiv:2602.04254*.

Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023. Codegen2: Lessons for training llms on programming and natural languages. *CoRR*, abs/2305.02309.

OpenAI. 2022. ChatML.

OpenAI. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. 2024. Training software engineering agents and verifiers with swe-gym. *arXiv preprint arXiv:2412.21139*.

Jingyu Peng, Maolin Wang, Nan Wang, Jiatong Li, Yuchen Li, Yuyang Ye, Wanyu Wang, Pengyue Jia, Kai Zhang, and Xiangyu Zhao. 2025. Logic jailbreak: Efficiently unlocking llm safety restrictions through formal logical expression. *arXiv preprint arXiv:2505.13527*.

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

Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, and 1 others. 2022. BLOOM: A 176B-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100*.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36:68539–68551.

Noam Shazeer. 2020. Glu variants improve transformer. *Preprint*, arXiv:2002.05202.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2020. Megatron-lm: Training multi-billion parameter language models using model parallelism. *Preprint*, arXiv:1909.08053.

KaShun Shum, Binyuan Hui, Jiawei Chen, Lei Zhang, Jiaxi Yang, Yuzhen Huang, Junyang Lin, Junxian He, and 1 others. 2025. Swe-rm: Execution-free feedback for software engineering agents. *arXiv preprint arXiv:2512.21919*.

Jinbo Su, Lingzhe Gao, Wei Li, Shihao Liu, Haojie Lei, Xinyi Wang, Yuanzhao Guo, Ke Wang, Daiting Shi, and Dawei Yin. 2025. Racqc: Advanced retrieval-augmented generation for chinese query correction. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 675–689.

Jinbo Su, Yuxuan Hu, Cuiping Li, Hong Chen, Jia Li, Lintao Ma, and Jing Zhang. 2026. Tablecache: Primary foreign key guided kv cache precomputation for low latency text-to-sql. *arXiv preprint arXiv:2601.08743*.

Yashar Talebirad and Amirhossein Nadiri. 2023. Multi-agent collaboration: Harnessing the power of intelligent llm agents. *arXiv preprint arXiv:2306.03314*.

Gemini Team. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *Preprint*, arXiv:2403.05530.

Qwen Team. Qwen3. 5: Towards native multimodal agents, february 2026. *URL https://qwen. ai/blog*.

Qwen Team. 2025. Qwen3-coder: Agentic coding in the world. *Blog post*.

Qwen team and etc. 2025. Qwen2.5 technical report. *Preprint*, arXiv:2412.15115.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, and 1 others. 2024a. Opendevin: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, and 1 others. 2024b. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I. Wang. 2025. Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution. *Preprint*, arXiv:2502.18449.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023a. Magicoder: Source code is all you need. *arXiv preprint arXiv:2312.02120*, abs/2312.02120.

Zeming Wei, Yifei Wang, Ang Li, Yichuan Mo, and Yisen Wang. 2023b. Jailbreak and guard aligned language models with only few in-context demonstrations. *arXiv preprint arXiv:2310.06387*.

Di Wu, Wasi Uddin Ahmad, Dejiao Zhang, Murali Krishna Ramanathan, and Xiaofei Ma. 2024. Repoformer: Selective retrieval for repository-level code completion. *arXiv preprint arXiv:2403.10059*.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying llm-based software engineering agents. *Preprint*, arXiv:2407.01489.

Yuan Xin, Dingfan Chen, Linyi Yang, Michael Backes, and Xiao Zhang. 2025. Jailbreaking attacks vs. content safety filters: How far are we in the llm safety arms race? *arXiv preprint arXiv:2512.24044*.

Shannan Yan, Jingchen Ni, Leqi Zheng, Jiajun Zhang, Peixi Wu, Dacheng Yin, Jing Lyu, Chun Yuan, and Fengyun Rao. 2026a. Adamem: Adaptive user-centric memory for long-horizon dialogue agents. *arXiv preprint arXiv:2603.16496*.

Shannan Yan, Leqi Zheng, Keyu Lv, Jingchen Ni, Hongyang Wei, Jiajun Zhang, Guangting Wang, Jing Lyu, Chun Yuan, and Fengyun Rao. 2026b. Learning cross-view object correspondence via cycle-consistent mask prediction. *arXiv preprint arXiv:2602.18996*.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 41 others. 2025. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

Jian Yang, Jiajun Zhang, Jiaxi Yang, Ke Jin, Lei Zhang, Qiyao Peng, Ken Deng, Yibo Miao, Tianyu Liu, Zeyu Cui, Binyuan Hui, and Junyang Lin. 2024a. Execrepobench: Multi-level executable code completion evaluation. *Preprint*, arXiv:2412.11990.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024b. Swe-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*.

Yang Yao, Xuan Tong, Ruofan Wang, Yixu Wang, Lujundong Li, Liang Liu, Yan Teng, and Yingchun Wang. 2025. A mousetrap: Fooling large reasoning models for jailbreak with chain of iterative chaos. *arXiv preprint arXiv:2502.15806*.

Sibo Yi, Yule Liu, Zhen Sun, Tianshuo Cong, Xinlei He, Jiaxing Song, Ke Xu, and Qi Li. 2024. Jailbreak attacks and defenses against large language models: A survey. *arXiv preprint arXiv:2407.04295*.

Zheng-Xin Yong, Cristina Menghini, and Stephen H Bach. 2023. Low-resource languages jailbreak gpt-4. *arXiv preprint arXiv:2310.02446*.

Youliang Yuan, Wenxiang Jiao, Wenxuan Wang, Jen-tse Huang, Pinjia He, Shuming Shi, and Zhaopeng Tu. 2023. Gpt-4 is too smart to be safe: Stealthy chat with llms via cipher. *arXiv preprint arXiv:2308.06463*.

Yi Zeng, Hongpeng Lin, Jingwen Zhang, Diyi Yang, Ruoxi Jia, and Weiyan Shi. 2024. How johnny can persuade llms to jailbreak them: Rethinking persuasion to challenge ai safety by humanizing llms. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 14322–14350.

Biao Zhang and Rico Sennrich. 2019. Root mean square layer normalization. *Preprint*, arXiv:1910.07467.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. Repocoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, pages 2471–2484. Association for Computational Linguistics.

Lei Zhang, Mouxiang Chen, Ruisheng Cao, Jiawei Chen, Fan Zhou, Yiheng Xu, Jiaxi Yang, Zeyao Ma, Liang Chen, Changwei Luo, and 1 others. 2026a. Megaflow: Large-scale distributed orchestration system for the agentic era. *arXiv preprint arXiv:2601.07526*.

Lei Zhang, Jiaxi Yang, Min Yang, Jian Yang, Mouxiang Chen, Jiajun Zhang, Zeyu Cui, Binyuan Hui, and Junyang Lin. 2025. Swe-flow: Synthesizing software engineering data in a test-driven manner. *Preprint*, arXiv:2506.09003.

Qianchi Zhang, Hainan Zhang, Liang Pang, Yongxin Tong, Hongwei Zheng, and Zhiming Zheng. 2026b. Less is more: Compact clue selection for efficient retrieval-augmented generation reasoning. In *Proceedings of the ACM Web Conference 2026*, pages 1971–1982.

Qianchi Zhang, Hainan Zhang, Liang Pang, Hongwei Zheng, and Zhiming Zheng. 2026c. Stable-rag: Mitigating retrieval-permutation-induced hallucinations in retrieval-augmented generation. *arXiv preprint arXiv:2601.02993*.

Heri Zhao, Jeffrey Hui, Joshua Howland, Nam Nguyen, Siqi Zuo, Andrea Hu, Christopher A. Choquette-Choo, Jingyue Shen, Joe Kelley, Kshitij Bansal, Luke Vilnis, Mateo Wirth, Paul Michel, Peter Choy, Pratik Joshi, Ravin Kumar, Sarmad Hashmi, Shubham Agrawal, Zhitao Gong, and 7 others. 2024. Codegemma: Open code models based on gemma. *CoRR*, abs/2406.11409.

Andy Zou, Zifan Wang, J Zico Kolter, and Matt Fredrikson. 2024. Universal and transferable adversarial attacks on aligned language models, 2023. *URL https://arxiv. org/abs/2307.15043*, 19.

Appendix

A Additional Related Works **15**
A.1 LLM Jailbreak 15
A.2 Code Agents 15
A.3 Code Editing 15

B Detailed Results **16**
B.1 Error Analysis: Gemini Series 16
B.2 Extended Performance Metrics 16

C Experiments Details **16**
C.1 Training Dataset 16
C.2 Benchmarks 16
C.3 Training Details 19
C.4 Evaluation Details 19
C.5 Additional Evaluation Details 20

D CrossCodeEval-Flex **21**

E Code Completion in Chat LLMs **21**
E.1 NL-FIM adaptations 21
E.2 Impracticality of Direct Code Generation 27

F Data Construction Details **28**
F.1 SRI Dataset Construction 28
F.2 General Instruction Data 28
F.3 Decontamination 28

G Design Decisions: Editable Region Sensitivity **28**

H Theoretical Analysis of Format Alignment **29**
H.1 Distributional Shift Hypothesis 29
H.2 Experiment: Format Perplexity Analysis 29
H.3 Results 30

A Additional Related Works

A.1 LLM Jailbreak

Jailbreaking techniques for large language models (LLMs) are broadly classified into white-box and black-box methods, distinguished by the attacker's level of access to model parameters (Yi et al., 2024; Ma et al., 2025; Zhang et al., 2026b,c; Xin et al., 2025). White-box approaches—such as gradient-based optimization (Zou et al., 2024; Peng et al., 2025), logits-driven methods (Guo et al., 2024b), and fine-tuning techniques—demonstrate high success rates but require full access to the model's internal states and parameters, limiting their practical applicability. In contrast, black-box methods operate without such access and typically rely on iterative querying. These attacks employ techniques such as template manipulation (e.g., scenario nesting (Ding et al., 2023a), context poisoning (Wei et al., 2023b; Hui et al., 2025; Chang et al., 2025; Luo et al., 2026), and code injection (Kang et al., 2024; Su et al., 2026, 2025)) and prompt transformation (e.g., cryptographic encoding (Deng et al., 2023), multilingual obfuscation (Yong et al., 2023), and evolutionary algorithms (Liu et al., 2023b)). Among these diverse strategies, prompt injection remains the predominant method for black-box attacks.

Black-box prompt injection can be further categorized into deterministic one-to-one attacks (Yuan et al., 2023; Handa et al., 2024) and probabilistic one-to-many attacks (Zeng et al., 2024). While early LLMs were susceptible to simple, rule-based injections, modern models with advanced reasoning capabilities have become more resilient to such straightforward attacks. To circumvent these defenses, researchers have developed multi-step reasoning chains that systematically exploit logical vulnerabilities in LLMs, enabling sophisticated attacks against even the most advanced systems (Yao et al., 2025). This vector of attack is particularly relevant in the domain of code completion, where models utilizing the FIM paradigm are highly susceptible to exploitation through malicious prefix or suffix code injection (Cheng et al., 2025).

A.2 Code Agents

Recent research highlights the pivotal role of large language models (LLMs) in the development of AI agents, showcasing their capability in facilitating complex task execution through tool utilization (Schick et al., 2023; Talebirad and Nadiri, 2023; Hong et al., 2023; **?**; Fang et al., 2026). Notable examples include ToolFormer, which enables tools to be used more effectively by LLMs; MetaGPT and BabyAGI, which demonstrate advancements in autonomous task management. Studies on self-edit and self-debug have further illustrated the capacity of code models to engage in multi-round interactions for code correction and improvement. Contemporary work also underscores the efficacy of agent systems like OpenDevin (Wang et al., 2024a) and SWE-Agent (Yang et al., 2024b) in handling complex programming tasks at the repository level, such as SWE-Bench (Jimenez et al., 2023; Pan et al., 2024; Jiang et al., 2026; Yan et al., 2026b,a; Ma et al., 2026; Chen et al., 2026; Zhang et al., 2026a; Shum et al., 2025). Additionally, Xia et al. (2024) introduced a lightweight agentless approach that achieves competitive performance in code editing tasks while maintaining architectural simplicity. Notably, these agent systems commonly adopt standard diff formats for communication with LLMs, demonstrating the format's effectiveness as a universal interface for code modification tasks.

A.3 Code Editing

Automated code editing has progressed from early sequence-to-sequence models that translated code as flat text(Chen et al., 2021b; Li et al., 2023), to more sophisticated structure-aware methods. These models operate on Abstract Syntax Trees (ASTs) or graphs to perform precise, local transformations (Chakraborty et al., 2022), but often ignore the broader context of a change. More recent work directly incorporates this context. Kim and Lee (2020) conditions its predictions on surrounding edits, while Liu et al. (2024) scales this concept to the repository level by locating and propagating changes across multiple files. This principle is particularly effective for large language models (LLMs), where providing "associated edits" as few-shot examples in prompts significantly improves performance without fine-tuning (Gupta et al., 2023). Concurrently, systems like Li et al. (2025) have focused on interpretability by learning and applying explicit edit rules derived from examples. However, many of these approaches were designed for specialized, non-LLM architectures and focus heavily on sub-tasks like edit localization. This makes them difficult to apply directly to the end-to-end inference process of modern chat-based LLMs.

B Detailed Results

In this section, we provide a comprehensive breakdown of our experimental results, including model specifications, detailed error analyses, and granular performance metrics across varying model scales.

B.1 Error Analysis: Gemini Series

A primary observation during our evaluation is the significant performance degradation of the Gemini model series (e.g., Gemini-1.5-Pro-Flash, Gemini-2.0-Flash) on code completion benchmarks when applying the SRI method. While these models possess strong general reasoning capabilities, they exhibit a specific failure mode in strict instruction following for formatting.

Further analysis reveals that the Gemini series struggles to adhere to the structural constraints of the SRI format. Specifically, the models frequently fail to generate the ‘SEARCH‘ block verbatim or omit the required `/* MIDDLE CODE TO COMPLETE */` identifier within the search block. We hypothesize that this behavior stems from the model overfitting to specific data formats (such as standard chat or function calling) during its post-training alignment phase, making it resistant to novel structural prompt engineering. Figure 6 presents a concrete example of this failure mode, where Gemini-1.5-Pro-Flash produces a search-and-replace block that entirely omits the crucial location identifier, rendering the replacement ambiguous.

B.2 Extended Performance Metrics

Table 5 lists the specific API versions and HuggingFace model codes used in our experiments. Tables 6 and 7 provide the complete performance comparisons on similarity-based and execution-based benchmarks, respectively. Additionally, to investigate the impact of model scaling on SRI effectiveness, Tables 8 and 9 detail the performance of the Qwen2.5-Coder series ranging from 0.5B to 32B parameters.

C Experiments Details

C.1 Training Dataset

Our training dataset consists of two main components: a novel Search-and-Replace Infilling (SRI) dataset and a conventional code instruction dataset. To ensure data integrity and prevent leakage, our training and benchmark datasets underwent a strict decontamination process, and we avoided any repository-level overlap between them. **SRI Data**. We constructed the SRI dataset using the open-source, repository-level dataset The Stack v2 (Lozhkov et al., 2024). To generate training samples, we first parse code files using tree-sitter to identify and extract fundamental logic blocks (e.g., functions, classes, loops). These blocks serve as the "middle" segments for our code infilling tasks. For each extracted block, we formulate a multi-level code completion query using its surrounding code as context. The ground truth is then formatted into a standard search-and-replace structure. This process yielded a large-scale dataset of 200K SRI samples, which we call SRI-200K, with a maximum context length of 32,768 tokens. From this dataset, we sampled a 20K subset to fine-tune our Qwen2.5-Coder-SRI models. For our ablation studies, we created a parallel version of this 20K subset where the ground truth was reformatted into a standard markdown code block, serving as the target output for natural language-based code completion prompts. **Instruction Data**. The second component is a general-purpose code instruction dataset. We sampled 60K question-answer pairs from the Glaive-Code-Assistant dataset (Glaive AI, 2023). To ensure a clear separation of tasks, we meticulously filtered this sample to remove all data related to code completion or search-and-replace tasks.

C.2 Benchmarks

To evaluate the code completion capabilities of the models, we selected five mainstream benchmarks originally designed for Fill-in-the-Middle (FIM) evaluation with base LLMs. Our selection includes three benchmarks based on similarity metrics including CrossCodeEval (Ding et al., 2023b), CrossCodeLongEval (Wu et al., 2024), and RepoEval (Zhang et al., 2023) and two based on unit test execution: SAFIM (Gong et al., 2024) and ExecRepoBench (Yang et al., 2024a). These benchmarks were chosen for their diversity and rigor. They cover a wide range of scenarios, including both repository-level and in-file completion tasks. Furthermore, their data is sourced from diverse origins, spanning popular GitHub repositories and algorithm code from Codeforces. This comprehensive set of benchmarks allows us to thoroughly assess the models' overall code completion performance under various conditions.

Table 5: Model code/API of the evaluated models

Model	Params	Model code/API
DeepSeek-V3-Base	671B-A37B	deepseek-ai/DeepSeek-V3-Base
DeepSeek-V3	671B-A37B	deepseek-ai/DeepSeek-V3
DeepSeek-V3-0324	671B-A37B	deepseek-ai/DeepSeek-V3-0324
DeepSeek-R1-0528	671B-A37B	deepseek-ai/DeepSeek-R1-0528
Claude-3.5-Haiku	-	claude-3-5-haiku-20241022
Claude-3.5-Sonnet	-	claude-3-5-sonnet-20241022
Claude-3.7-Sonnet	-	claude-3-7-sonnet-20250219
Claude-3.7-Sonnet-think	-	claude-3-7-sonnet-20250219-thinking
Claude-4-Sonnet	-	claude-4-sonnet-20250514
GPT-4o-1120	-	gpt-4o-2024-11-20
GPT-4o-0806	-	gpt-4o-2024-08-06
GPT-4o-mini	-	gpt-4o-mini-2024-08-17
GPT-4.1	-	gpt-4.1-2025-04-14
o1	-	o1-2024-12-17
o3-mini	-	o3-mini-2025-01-31
Gemini-1.5-Pro-Flash	-	gemini-1.5-pro-flash
Gemini-1.5-Pro	-	gemini-1.5-pro
Gemini-2.0-Flash	-	gemini-2.0-flash
Gemini-2.0-Flash-Think	-	gemini-2.0-flash-thinking
Grok-3	-	grok-3
Qwen2.5-Coder-32B	32B	Qwen/Qwen2.5-Coder-32B
Qwen2.5-Coder-32B-Instruct	32B	Qwen/Qwen2.5-Coder-32B-Instruct
Qwen3-30B-A3B	30B-A3B	Qwen/Qwen3-30B-A3B
Qwen3-235B-A22B	235B-A22B	Qwen/Qwen3-235B-A22B
Qwen3-Coder-480B-A35B	480B-A35B	Qwen/Qwen3-Coder-480B-A35B-Instruct

Error search replace case for Gemini

````
```replace
<<<<<<< SEARCH
    """
    Preprocess a mask for the model.
    """
    mask = mask.convert("L")
    w, h = mask.size
    w, h = map(lambda x: x - x \% 32, (w, h))  # resize to
       integer multiple of 32
    mask = mask.resize(
        (w // scale_factor, h // scale_factor), resample=PIL.
           Image.NEAREST
    )
    mask = np.array(mask).astype(np.float32) / 255.0
    mask = np.tile(mask, (4, 1, 1))
    mask = mask[None].transpose(0, 1, 2, 3)  # what does this
       step do?
    mask = 1 - mask  # repaint white, keep black
    mask = torch.from_numpy(mask)
    return mask
=======
    """
    Preprocess a mask for the model.
    """
    mask = mask.convert("L")
    w, h = mask.size
    w, h = map(lambda x: x - x % 32, (w, h))  # resize to
       integer multiple of 32
    mask = mask.resize(
        (w // scale_factor, h // scale_factor), resample=PIL.
           Image.NEAREST
    )
    mask = np.array(mask).astype(np.float32) / 255.0
    mask = np.tile(mask, (4, 1, 1))
    mask = mask[None].transpose(0, 1, 2, 3)  # what does this
       step do?
    mask = 1 - mask  # repaint white, keep black
    mask = torch.from_numpy(mask)
    return mask

>>>>>>> REPLACE
```
````

Figure 6: Error search replace case for Gemini

Model	CrossCodeEval				RepoEval				CrossCodeLongEval			
	EM		*ES*		*EM*		*ES*		*EM*		*ES*	
	FIM	**SRI**	**FIM**	**SRI**	**FIM**	**SRI**	**FIM**	**SRI**	**FIM**	**SRI**	**FIM**	**SRI**
Base Models												
Qwen2.5-Coder-32B	57.1	-	86.8	-	51.6	-	78.5	-	36.8	-	66.4	-
DeepSeek-V3-Base	61.9	-	88.5	-	63.6	-	86.2	-	50.5	-	77.9	-
Non-reasoning Chat Models												
DeepSeek-V3	39.0	56.1 ↑17.1	70.3	86.2 ↑15.9	37.7	54.3 ↑16.6	60.4	78.9 ↑18.5	25.4	34.8 ↑9.4	53.5	67.4 ↑13.9
DeepSeek-V3-0324	44.5	56.0 ↑11.5	80.0	85.7 ↑5.7	45.8	54.5 ↑8.7	68.2	79.3 ↑11.1	30.9	34.8 ↑3.9	60.2	68.2 ↑8.0
Claude-3.5-Haiku	23.9	44.5 ↑20.6	73.2	75.2 ↑2.0	31.9	44.4 ↑12.5	58.2	67.8 ↑9.6	17.7	27.1 ↑9.4	49.8	56.2 ↑6.4
Claude-3.5-Sonnet	47.5	56.0 ↑8.5	82.3	85.6 ↑3.3	48.0	54.4 ↑6.4	72.1	80.2 ↑8.1	32.4	36.3 ↑3.9	63.2	69.4 ↑6.2
Claude-3.7-Sonnet	48.3	56.7 ↑8.4	81.6	85.7 ↑4.1	49.3	55.0 ↑5.7	73.6	80.2 ↑6.6	32.5	38.2 ↑5.7	64.0	70.5 ↑6.5
Claude-4-Sonnet	57.5	60.8 ↑3.3	87.3	87.2 ↓0.1	55.7	58.9 ↑3.2	78.7	82.0 ↑3.3	38.1	38.9 ↑0.7	68.8	69.5 ↑0.7
GPT-4o-1120	34.2	44.1 ↑9.9	72.5	80.9 ↑8.4	25.7	45.0 ↑19.3	52.1	72.6 ↑20.5	21.7	23.6 ↑1.9	51.6	56.4 ↑4.8
GPT-4o-0806	40.3	41.7 ↑1.4	77.3	79.5 ↑2.2	37.2	46.8 ↑9.6	61.5	73.7 ↑12.2	20.6	26.4 ↑5.8	50.7	58.2 ↑7.5
GPT-4o-mini	9.0	27.7 ↑18.7	59.6	68.3 ↑8.7	19.6	27.7 ↑8.1	48.0	54.6 ↑6.6	12.1	11.1 ↓1.0	41.3	39.6 ↓1.7
GPT-4.1	46.1	45.8 ↓0.3	81.0	79.8 ↓1.2	47.0	46.9 ↓0.1	71.7	73.2 ↑1.5	30.8	28.5 ↓2.3	62.2	61.2 ↓1.0
Gemini-1.5-Pro-Flash	42.7	30.9 ↓11.8	80.2	58.7 ↓21.5	42.3	37.6 ↓4.7	70.6	60.3 ↓10.3	26.0	9.8 ↓16.2	57.5	27.9 ↓29.6
Gemini-1.5-Pro	49.6	40.8 ↓8.8	83.3	68.6 ↓14.7	50.4	44.1 ↓6.3	75.8	69.3 ↓6.5	33.7	28.3 ↓5.4	64.4	57.6 ↓6.8
Gemini-2.0-Flash	43.8	6.8 ↓37.0	75.7	16.5 ↓59.2	44.4	9.0 ↓35.4	67.2	15.1 ↓52.1	29.0	4.4 ↓24.6	56.8	14.3 ↓42.5
Qwen3-Coder-480B-A35B	56.1	**62.9** ↑6.9	85.8	**89.5** ↑3.7	51.1	**65.1** ↑14.0	73.2	**85.4** ↑12.2	36.2	**45.9** ↑9.7	65.3	**74.7** ↑9.4
Qwen2.5-Coder-32B-Instruct	23.4	44.4 ↑21.0	69.9	76.8 ↑6.9	31.1	44.8 ↑13.7	56.1	69.7 ↑13.6	20.4	25.9 ↑5.5	50.3	55.8 ↑5.5
SRI-Coder-32B	11.3	57.6 ↑46.3	37.9	87.3 ↑49.3	17.3	54.4 ↑37.1	37.3	78.9 ↑41.5	14.6	37.1 ↑22.5	33.2	67.9 ↑34.7
Reasoning Chat Models												
DeepSeek-R1-0528	44.6	51.1 ↑6.5	82.3	84.0 ↑1.7	48.1	49.2 ↑1.1	75.1	76.0 ↑0.9	29.2	29.7 ↑0.5	62.2	62.6 ↑0.4
Claude-3.7-Sonnet-think	49.4	**54.6** ↑5.2	82.5	**84.5** ↑2.0	9.1	53.0 ↑43.9	27.6	78.0 ↑50.4	30.6	34.9 ↑4.3	60.7	66.9 ↑6.2
Gemini-2.0-Flash-think	37.4	32.9 ↓4.5	61.3	62.3 ↑1.0	46.3	44.8 ↓1.5	68.9	64.9 ↓4.0	28.9	21.2 ↓7.7	56.4	47.4 ↓9.0
Grok-3	46.4	54.4 ↑8.0	82.1	84.1 ↑2.0	50.1	**53.6** ↑3.5	75.8	**78.7** ↑2.9	32.4	**35.2** ↑2.8	65.2	**68.0** ↑2.8
o1-2024-12-17	26.9	42.7 ↑15.8	75.4	78.4 ↑3.0	44.5	48.3 ↑3.8	68.4	73.6 ↑5.2	26.6	24.4 ↓2.2	56.6	57.0 ↑0.4
o3-mini	23.7	27.6 ↑3.9	39.1	69.9 ↑30.8	26.4	39.1 ↑12.7	37.1	61.4 ↑24.3	9.7	20.9 ↑11.2	21.1	48.9 ↑27.8
Qwen3-30B-A3B	13.7	36.1 ↑22.4	59.7	71.5 ↑11.8	20.0	33.6 ↑13.6	44.3	61.2 ↑16.9	12.9	17.6 ↑4.7	36.7	45.1 ↑8.4
Qwen3-235B-A22B	14.6	42.7 ↑28.2	35.6	76.8 ↑41.3	19.1	42.7 ↑23.6	30.6	68.9 ↑38.3	15.0	23.2 ↑8.2	32.5	53.5 ↑21.0

Table 6: Performance comparison on similarity-based benchmarks (CrossCodeEval, RepoEval, and CrossCodeLongEval). For base models, **FIM** denotes special token based fill-in-the-middle code completion, for chat models, **FIM** denotes natural language-based code completion prompting, while **SRI** represents our method. **Bold** and underlined values indicate the best and second-best performance respectively within the model type. ↑n and ↓n indicate performance improvements and degradations of SRI compared to FIM.

C.3 Training Details

All models were trained on 16 NVIDIA A100 (80GB) GPUs using the Megatron-LM framework(Shoeybi et al., 2020). We configured the training with a context length of 32,768 tokens, a global batch size of 256, and a micro-batch size of 1. The learning rate was set to 5×10^{-5}, with a warm-up phase of 30 steps and a minimum learning rate of 5×10^{-6}, followed by a linear decay over 853 steps. We used BFloat16(BF16) for mixed-precision training. The model architecture employs the SwiGLU (Shazeer, 2020) activation function and RMSNorm (Zhang and Sennrich, 2019) (with $\epsilon = 10^{-6}$) for normalization. Gradient clipping was set to 1.0, and a weight decay of 0.1 was applied. The total training dataset comprised 80K samples: 20K for our SRI code completion task, 60K for general code-related instruction following, and 100 samples for safety alignment. To avoid potential data contamination, we fine-tuned our models starting from the pre-trained Qwen2.5-Coder base models rather than the instruction-tuned Qwen2.5-Coder-Instruct series. This decision prevents any pre-existing exposure to code completion or search-and-replace SFT data in the instruction-tuned models from confounding our experimental results. For further details on the model architecture, please refer to Hui et al. (2024).

C.4 Evaluation Details

Our evaluation encompasses a diverse set of models, including the GPT series (OpenAI, 2023), Claude series (Anthropic, 2023a), Gemini series (Team, 2024), DeepSeek-V3 (DeepSeek-AI and etc., 2024), DeepSeek-R1 (DeepSeek-AI et al., 2025), Qwen2.5 series (team and etc., 2025),

Model	ExecRepoBench		SAFIM									
	Pass@1		*Block*		*Block V2*		*API*		*Control*		*Average*	
	FIM	**SRI**	**FIM**	**SRI**	**FIM**	**SRI**	**FIM**	**SRI**	**FIM**	**SRI**	**FIM**	**SRI**
Base Models												
Qwen2.5-Coder-32B	25.7	-	62.9	-	65.2	-	75.5	-	76.4	-	70.0	-
DeepSeek-V3-Base	36.5	-	76.3	-	77.9	-	79.4	-	84.8	-	79.6	-
Non-reasoning Chat Models												
DeepSeek-V3	35.6	61.8 ↑26.2	60.5	**70.1** ↑9.6	59.7	70.6 ↑10.9	55.8	73.6 ↑17.8	64.1	80.6 ↑16.5	60.0	73.7 ↑13.7
DeepSeek-v3-0324	36.9	65.5 ↑28.6	53.3	69.8 ↑16.5	56.5	71.1 ↑14.6	68.1	74.5 ↑6.4	65.3	**81.8** ↑16.5	60.8	74.3 ↑13.5
Claude-3.5-Haiku	35.6	57.7 ↑22.1	60.1	66.3 ↑6.2	59.7	68.0 ↑8.3	65.2	71.6 ↑6.4	68.5	77.4 ↑8.9	63.4	70.8 ↑7.4
Claude-3.5-Sonnet	41.8	66.2 ↑24.4	60.6	65.9 ↑5.3	60.2	68.0 ↑7.8	61.3	64.2 ↑2.9	68.9	75.2 ↑6.3	62.7	68.3 ↑5.6
Claude-3.7-Sonnet	42.0	62.8 ↑20.8	61.6	67.8 ↑6.2	60.9	70.5 ↑9.6	64.2	66.1 ↑1.9	66.1	79.3 ↑13.2	63.2	70.9 ↑7.7
Claude-4-Sonnet	61.2	67.1 ↑5.9	67.3	69.8 ↑2.4	69.2	**72.9** ↑3.7	72.9	74.2 ↑1.3	78.6	80.6 ↑2.0	72.0	**74.4** ↑2.4
GPT-4o-1120	36.2	55.9 ↑19.7	59.5	62.4 ↑2.9	55.9	62.3 ↑6.4	65.2	68.1 ↑2.9	58.6	75.4 ↑16.8	59.8	67.0 ↑7.2
GPT-4o-0806	39.4	58.4 ↑19.0	47.9	59.7 ↑11.8	46.6	57.0 ↑10.4	64.2	69.4 ↑5.2	54.9	67.3 ↑12.4	53.4	63.4 ↑10.0
GPT-4o-mini	28.0	40.8 ↑12.8	25.2	32.4 ↑7.2	26.8	32.4 ↑5.6	23.9	44.8 ↑20.9	9.5	43.4 ↑33.9	21.3	38.2 ↑16.9
GPT-4.1	41.4	59.6 ↑18.2	51.2	66.6 ↑15.4	53.1	63.5 ↑10.4	68.1	71.3 ↑3.2	59.8	68.6 ↑8.8	58.0	67.5 ↑9.5
Gemini-1.5-Pro-Flash	32.7	45.8 ↑13.1	50.0	30.5 ↓19.5	49.6	32.3 ↓17.3	65.2	54.5 ↓10.7	51.1	33.9 ↓17.2	54.0	37.8 ↓16.2
Gemini-1.5-Pro	37.9	53.4 ↑15.5	60.8	56.0 ↓4.8	56.3	53.2 ↓3.1	69.4	67.4 ↓2.0	57.7	56.7 ↓1.0	61.0	58.3 ↓2.7
Gemini-2.0-Flash	41.1	31.1 ↓10.0	62.2	19.7 ↓42.5	58.0	17.6 ↓40.4	66.8	24.2 ↓42.6	64.1	17.8 ↓46.3	62.8	19.8 ↓43.0
Qwen3-Coder-480B-A22B	52.1	**79.1** ↑27.0	63.4	66.3 ↑2.9	65.5	69.6 ↑4.1	74.8	**78.1** ↑3.2	74.6	81.2 ↑6.6	69.6	73.8 ↑4.2
Qwen2.5-Coder-32B-Instruct	43.6	46.0 ↑2.4	36.8	44.9 ↑8.1	37.8	48.6 ↑10.8	48.4	68.4 ↑20.0	31.1	55.8 ↑24.7	38.5	54.4 ↑15.9
SRI-Coder-32B	24.5	61.6 ↑37.1	10.3	60.5 ↑50.1	15.3	64.5 ↑49.2	5.3	74.9 ↑69.6	8.0	75.6 ↑67.6	9.8	69.9 ↑60.1
Reasoning Chat Models												
DeepSeek-R1-0528	42.1	60.1 ↑18.0	62.3	69.8 ↑7.5	60.5	**72.3** ↑11.8	57.1	**75.9** ↑18.8	66.2	**79.8** ↑13.6	61.5	**74.5** ↑13.0
Claude-3.7-Sonnet-think	44.2	61.3 ↑17.1	64.3	68.3 ↑4.0	63.6	70.7 ↑7.1	58.4	66.8 ↑8.4	72.9	79.2 ↑6.3	64.8	71.3 ↑6.5
Gemini-2.0-Flash-think	42.7	42.2 ↓0.5	35.2	33.7 ↓1.5	36.1	34.0 ↓2.1	63.6	67.7 ↑4.1	43.7	38.5 ↓5.2	44.6	43.5 ↓1.1
Grok-3	39.1	**63.4** ↑24.3	58.7	63.3 ↑4.6	59.1	66.6 ↑7.5	61.9	72.3 ↑10.4	65.5	73.8 ↑8.3	61.3	69.0 ↑7.7
o1-2024-12-17	41.1	51.5 ↑10.4	62.6	68.7 ↑6.1	65.9	71.1 ↑5.2	67.1	72.2 ↑5.1	65.9	74.4 ↑8.5	65.4	71.6 ↑6.2
o3-mini	43.0	57.0 ↑14.0	32.1	**76.4** ↑44.3	28.0	71.6 ↑43.6	44.8	57.7 ↑12.9	38.2	73.1 ↑34.9	35.8	69.7 ↑34.1
Qwen3-30B-A3B	33.0	48.5 ↑15.5	29.9	42.6 ↑12.7	34.7	48.0 ↑13.3	50.0	64.8 ↑14.8	34.1	58.3 ↑24.2	37.2	53.4 ↑16.2
Qwen3-235B-A22B	35.6	54.7 ↑19.1	33.3	47.5 ↑12.6	36.1	48.7 ↑12.6	35.2	63.5 ↑28.4	52.6	58.1 ↑5.5	39.3	54.4 ↑15.2

Table 7: Performance comparison on unit tests-based benchmarks (ExecRepoBench and SAFIM). In SAFIM, all metrics are reported as Pass@1 except API which uses Exact Match (EM).

Qwen3 series (Yang et al., 2025). For closed-source commercial models, we utilized their official paid APIs. The complete evaluation across all five benchmarks consumes approximately 180M input tokens per model, with varying costs due to different API pricing structures. Open-source models were deployed locally using the vLLM serving framework. To ensure a fair comparison of inference efficiency, all Qwen2.5-Coder models were served on 2 NVIDIA A100 GPUs with a tensor parallelism (TP) of 2. All inference requests were formatted using the standard OpenAI API structure. We employed greedy decoding, set the presence penalty to 0, and limited the maximum output to 256 tokens.

We configured context lengths according to each benchmark's standard: the full context for SAFIM (Gong et al., 2024), 32k tokens for ExecRepoBench (Yang et al., 2024a), and 8k tokens for CrossCodeEval (Ding et al., 2023b), RepoEval (Zhang et al., 2023), and CrossCodeLongEval (Wu et al., 2024). Crucially, to ensure a fair and controlled comparison, two principles were strictly followed. First, for all three inference methods, the provided prefix, suffix, and cross-file contexts were kept identical and perfectly aligned with the original datasets. Second, to equalize the amount of information provided to the model—since FIM and natural language prompts implicitly signal the location of the missing code—we introduced a `/* MIDDLE CODE TO COMPLETE */` identifier within the SRI prompt to explicitly mark the target completion location.

C.5 Additional Evaluation Details

Evaluating the raw output of our SRI method presents a challenge, as the search-and-replace format is not directly compatible with standard code completion metrics like Exact Match (EM) and Edit Similarity (ES). To bridge this gap, we first post-process the generated output to isolate only the newly inserted middle code. This extraction is per-

Model	CrossCodeEval				RepoEval				CrossCodeLongEval			
	EM		*ES*		*EM*		*ES*		*EM*		*ES*	
	FIM	**SRI**	**FIM**	**SRI**	**FIM**	**SRI**	**FIM**	**SRI**	**FIM**	**SRI**	**FIM**	**SRI**
Qwen2.5-Coder-0.5B	24.6	-	68.9	-	28.1	-	63.0	-	19.7	-	51.1	-
SRI-Coder-0.5B	-	24.4	-	66.5	-	25.2	-	57.3	-	16.4	-	48.1
Qwen2.5-Coder-1.5B	40.2	-	78.2	-	40.5	-	71.7	-	28.3	-	59.2	-
SRI-Coder-1.5B	-	39.7	-	76.6	-	39.4	-	68.8	-	29.2	-	60.5
Qwen2.5-Coder-3B	44.9	-	80.9	-	44.0	-	74.0	-	30.1	-	61.3	-
SRI-Coder-3B	-	43.5	-	79.8	-	44.3	-	74.6	-	30.9	-	61.9
Qwen2.5-Coder-7B	49.3	-	83.1	-	46.3	-	75.1	-	33.4	-	63.9	-
SRI-Coder-7B	-	50.3	-	84.0	-	46.5	-	74.0	-	34.5	-	62.2
Qwen2.5-Coder-14B	55.4	-	86.1	-	50.6	-	79.0	-	36.2	-	65.8	-
SRI-Coder-14B	-	53.3	-	85.5	-	52.4	-	78.2	-	34.0	-	66.2
Qwen2.5-Coder-32B	57.1	-	86.8	-	51.6	-	78.5	-	36.9	-	66.5	-
SRI-Coder-32B	-	57.6	-	87.1	-	54.4	-	78.9	-	37.1	-	67.9

Table 8: Performance of Qwen2.5-Coder models with and without SRI on various benchmarks. For each model pair, the first row shows FIM performance, and the second row shows SRI performance.

Model	ExecRepoBench		SAFIM									
	Pass@1		*Block*		*Block V2*		*API*		*Control*		*Average*	
	FIM	**SRI**	**FIM**	**SRI**	**FIM**	**SRI**	**FIM**	**SRI**	**FIM**	**SRI**	**FIM**	**SRI**
Qwen2.5-Coder-0.5B	22.0	-	24.6	-	30.5	-	47.7	-	37.2	-	35.0	-
SRI-Coder-0.5B	-	36.8	-	15.0	-	20.3	-	41.9	-	23.8	-	25.3
Qwen2.5-Coder-1.5B	17.2	-	38.9	-	44.8	-	64.8	-	52.6	-	50.3	-
SRI-Coder-1.5B	-	46.0	-	33.3	-	39.8	-	64.4	-	48.2	-	46.4
Qwen2.5-Coder-3B	22.3	-	46.6	-	50.3	-	65.8	-	60.3	-	55.8	-
SRI-Coder-3B	-	48.6	-	43.3	-	53.9	-	68.5	-	57.7	-	55.9
Qwen2.5-Coder-7B	19.8	-	52.2	-	56.8	-	70.6	-	65.2	-	61.2	-
SRI-Coder-7B	-	51.5	-	44.6	-	50.6	-	69.1	-	61.7	-	57.0
Qwen2.5-Coder-14B	22.6	-	58.8	-	62.5	-	74.5	-	72.3	-	67.0	-
SRI-Coder-14B	-	59.7	-	58.1	-	63.6	-	73.0	-	66.0	-	67.2
Qwen2.5-Coder-32B	25.7	-	62.9	-	65.2	-	75.5	-	76.4	-	70.0	-
SRI-Coder-32B	-	61.6	-	60.5	-	64.5	-	74.9	-	75.6	-	69.9

Table 9: Performance comparison of Qwen2.5-Coder models on unit test-based benchmarks (ExecRepoBench and SAFIM). For each model size, FIM and SRI performances are shown in separate rows.

formed reliably using the `extract_replace_code` function, detailed in Figure 7 and 8.

Table 5 details all models used in our experiments to facilitate the reproducibility of our findings. For the MBPP and HumanEval benchmarks, we used the versions provided by EvalPlus (Liu et al., 2023a), which augment the original datasets with additional test cases for more rigorous evaluation.

Table 8 and 9 details all SRI-Coder series models' performance on code completion benchmarks.

D CrossCodeEval-Flex

In this section, we show some samples in our CrossCodeEval-Flex benchmark in Figure 9 and 10, the code between two @ is perturbated.

E Code Completion in Chat LLMs

E.1 NL-FIM adaptations

To comprehensively evaluate the adaptability of Chat LLMs, we investigate three distinct natural language prompting strategies for FIM tasks. **Standard FIM Completion** (Figure 11) employs direct directives, explicitly instructing the model to generate the code necessary to bridge the provided prefix and suffix. In contrast, **Dialogue-Based FIM Reconstruction** (Figure 12) frames the task as a conversational repair scenario, where the user simulates a request for assistance in restoring missing logic. Finally, **Template-Based FIM Infilling** (Figure 13) utilizes a rigid, slot-filling structure to demarcate context boundaries, relying on the model's pattern recognition to identify and complete the target region.

```python
import re

def extract_replace_code(text: str) -> str:
    """
    Extracts the differential code part from text in a search/replace format,
    handling inline completion.

    Args:
        text (str): The text string containing the search/replace format.

    Returns:
        str: The extracted code to be completed.
    """
    # Find the SEARCH and REPLACE parts
    search_pattern = r"<{2,}\s*SEARCH\n(.*?)\n\s*={3,}"
    replace_pattern = r"={3,}\n(.*?)\n\s*>{2,}\s*REPLACE"

    search_match = re.search(search_pattern, text, re.DOTALL)
    replace_match = re.search(replace_pattern, text, re.DOTALL)

    if not replace_match:
        return ""
    if not search_match:
        return replace_match.group(1)

    search_code = search_match.group(1).strip()
    replace_code = replace_match.group(1).strip()

    # Find the position of the middle marker
    middle_marker = "/* MIDDLE_CODE_TO_COMPLETE */"
    if middle_marker not in search_code:
        # Handle the inline completion case
        if search_code and replace_code:
            # If search_code is a prefix of replace_code, it means the latter part needs completion.
            if replace_code.startswith(search_code):
                return replace_code[len(search_code) :]
            # If search_code is a suffix of replace_code, it means the beginning  part needs completion.
            elif replace_code.endswith(search_code):
                return replace_code[: -len(search_code)]

            # Find the longest common prefix
            i = 0
            while i < len(search_code) and i < len(replace_code) and search_code[i] == replace_code[i]:
                i += 1

            # Find the longest common suffix
            j = 1
            while j <= len(search_code) and j <= len(replace_code) and search_code[-j] == replace_code[-j]:
                j += 1
            j -= 1  # Step back one position because the loop checked one extra time

            # Extract the middle differential part
            if j > 0:
                return replace_code[i:-j]
            return replace_code[i:]

        # If search_code is empty or completely different from replace_code, return the full replace_code
        return replace_code

    # Split the search code into two parts: before and after the marker
    before_middle, after_middle = search_code.split(middle_marker)
    before_middle = before_middle.rstrip("\n")
    after_middle = after_middle.lstrip("\n")
```

Figure 7: Extract Replace Code Function (part 1)

```
1  # Handle special cases: marker is on the first or last line
2      if not before_middle.strip():  # marker is on the first line
3          return replace_code.split(after_middle)[0].strip("\n") if after_middle
           ↪ else replace_code.strip("\n")
4
5      if not after_middle.strip():  # marker is on the last line
6          return replace_code[len(before_middle) :].strip("\n") if before_middle
           ↪ else replace_code.strip("\n")
7
8      # Start with the full replace code and trim the identical parts
9      result = replace_code
10
11     # If the 'before' part is not empty, remove it as a prefix
12     if before_middle:
13         if result.startswith(before_middle):
14             result = result[len(before_middle) :].lstrip("\n")
15         else:
16             # Handle the inline completion case
17             last_line_before = before_middle.split("\n")[-1]
18             if result.startswith(last_line_before):
19                 # Find the position of the first differing character
20                 i = 0
21                 while i < len(last_line_before) and i < len(result) and
                       ↪ last_line_before[i] == result[i]:
22                     i += 1
23                 result = result[i:]
24
25     # If the 'after' part is not empty, remove it as a suffix
26     if after_middle:
27         if result.endswith(after_middle):
28             result = result[: -len(after_middle)].rstrip("\n")
29         else:
30             # Handle the inline completion case
31             first_line_after = after_middle.split("\n")[0]
32             if result.endswith(first_line_after):
33                 # Find the position of the last differing character
34                 i = -1
35                 while -i <= len(first_line_after) and -i <= len(result) and
                       ↪ first_line_after[i] == result[i]:
36                     i -= 1
37                 result = result[: i + 1]
38
39     return result.strip("\n")
```

Figure 8: Extract Replace Code Function (part 2)

```
1          public List<BaseRule> GetRules()
2          {
3              List<BaseRule> rules = new List<BaseRule>();
4              @ .dncCo()rPa;grRsn(emce.RsgFeahi rnlue)o @
5              rules.AddRange(managerConfig.FolderRules);
6
7              return rules;
8          }
9
10         public Dictionary<string,
11         /* MIDDLE CODE TO COMPLETE */
12
13                     Dictionary<string, LassoProfile> lassoProfiles = new
                           ↪ Dictionary<string, LassoProfile>();
14
15             // Load lasso profiles
16             @ foreach (profile var in managerConfig.Profiles) @
17             {
18                 if (!lassoProfiles.ContainsKey(profile.Name))
19                 {
20                     lassoProfiles.Add(profile.Name, profile);
21                 }
22             }
23
24             return lassoProfiles;
25         }
26
27         private string GetConfigFilePath()
28             => Path.Combine(Path.GetDirectoryName(Assembly.GetExecutingAssembly
                   ↪ ().Location), ConfigFileName);
29     }
30 }
```

Figure 9: CrossCodeEval-Flex case 1

```
1  package io.github.aaronai;
2
3  import io.github.aaronai.http.HttpClientUtil;
4  import io.github.aaronai.mq.RocketMqClients;
5  import io.opentelemetry.api.GlobalOpenTelemetry;
6  import io.opentelemetry.api.OpenTelemetry;
7  import io.opentelemetry.api.trace.Span;
8  import io.opentelemetry.api.trace.Tracer;
9  import org.apache.rocketmq.client.apis.ClientException;
10 import org.apache.rocketmq.client.apis.consumer.ConsumeResult;
11 import org.slf4j.Logger;
12 import org.slf4j.LoggerFactory;
13
14 public class ProcessC {
15     private static final Logger logger = LoggerFactory.getLogger(ProcessC.class)
           ↪ ;
16
17     @SuppressWarnings("resource")
18     public static void main(String[] args) throws ClientException {
19         RocketMqClients.CreatePushConsumer(messageView -> {
20             @ g"ei}I,c(s.i(eVd{=aevmg)oe,;r"s g)ss.m eeMewt s s hgimeetdLaesrCmo
                 ↪  nfeagv); @
21         /* MIDDLE CODE TO COMPLETE */
22
23             final OpenTelemetry openTelemetry = GlobalOpenTelemetry.get();
24             @ r"tr)ec=pTihb.oar.aTrne;"naa.tl(to yl.n e i reefrt mCg i agetioG @
25             final Span span = tracer.spanBuilder("ExampleDownstreamSpan").
                 ↪ startSpan();
26             try {
27                 // do something here.
28                 Thread.sleep(1000);
29             } catch (InterruptedException ignored) {
30             }
31             span.end();
32
33             return ConsumeResult.SUCCESS;
34         });
35     }
36 }
```

Figure 10: CrossCodeEval-Flex case 2

Standard FIM Completion

You are a code completion assistant. Your
based on the provided context code, prefix
code, and suffix code.

Requirements:
between the prefix and suffix
2. Do not repeat or modify any existing
code from the context, prefix, or suffix
3. Format your response within a code
block
4. Maintain consistent indentation with the
surrounding code
5. Do not modify any other code
6. Do not add any other text or comments

The input will be provided in the following
##Context Code##: [Full context code if
any]
##Prefix Code##: [Code before the missing
part]
##Suffix Code##: [Code after the missing
part]

Figure 11: The **Standard FIM Completion** prompt format. This approach uses explicit natural language instructions to define the prefix and suffix, directly commanding the model to generate the connecting code segment.

Dialogue-Based FIM Reconstruction

You are participating in a code review
a gap in the middle, and you need to help
reconstruct what's missing

The developer will provide:
code structure
The code written BEFORE the gap (prefix)
The code written AFTER the gap (suffix)

Your role:
Determine what logic/code should bridge
them
Respond with ONLY the bridging code
segment
Preserve exact indentation and style
Don't alter any provided code
Don't include explanations or markdown
formatting outside the code block

Input structure:
context]
##Prefix Code##: [Code ending before gap]
##Suffix Code##: [Code starting after gap]

Figure 12: The **Dialogue-Based FIM Reconstruction** prompt format. The task is simulated as a conversational interaction where the user describes a "missing code" scenario, prompting the model to act as an assistant and restore the lost logic.

Template-Based FIM Infilling

You are a code template processor. You
a section has been marked for automatic
generation. Your job is to fill in the marked
section.

Template structure you'll receive:
class context]
##Prefix Code##: [Everything written up to
the insertion point]
##Suffix Code##: [Everything written after
the insertion point]

Processing rules:
gap
\ Match indentation level exactly as
shown in prefix/suffix
\ Treat prefix and suffix as immutable
never modify them
\ No additional commentary or formatting
\ Ensure syntactic continuity from prefix
\ your code \ suffix
\ Wrap output in a code block

Think of this as: PREFIX +
YOUR GENERATION
+ SUFFIX = Complete Code

Figure 13: The **Template-Based FIM Infilling** prompt format. This method utilizes a fixed textual template to structurally isolate the code context, guiding the model to perform a slot-filling task within the defined boundaries.

E.2 Impracticality of Direct Code Generation

The challenge of generating a correct and complete code sequence S^* from a natural language prompt C is inherent to the autoregressive nature of Large Language Models (LLMs). The joint probability of generating a sequence $S = (w_1, \dots, w_L)$ is factorized as:

$$P(S|C) = \prod_{t=1}^{L} P(w_t|C, w_1, \dots, w_{t-1}) \quad (3)$$

where each conditional probability $P(w_t|\cdot)$ is derived from the model's internal state, which is highly sensitive to the entire preceding context.

Let us denote the ideal, unambiguous prompt that perfectly specifies a target sequence S^* as C^*. In practice, a user-provided prompt C is often an imprecise approximation of C^*. This initial imprecision introduces a divergence in the model's internal representation, such that the hidden state $\mathbf{h}(C) \neq \mathbf{h}(C^*)$. This immediately results in a distributional shift for the first generated token, quantified by the Kullback-Leibler (KL) divergence:

$$D_{KL}\big(P(\cdot|C^*) \| P(\cdot|C)\big) > 0 \quad (4)$$

Consequently, the probability of generating the correct first token w_1^* is diminished, i.e., $P(w_1^*|C) < P(w_1^*|C^*)$.

This initial error compounds at each subsequent step of the generation process. The probability of generating the target sequence S^* given the actual prompt C is the probability of staying on the "golden path" of correct tokens:

$$P(S^*|C) = \prod_{t=1}^{L} P(w_t^*|C, w_1^*, \dots, w_{t-1}^*) \quad (5)$$

At any step t, the context for generation $(C, w_1^*, \dots, w_{t-1}^*)$ differs from the ideal context $(C^*, w_1^*, \dots, w_{t-1}^*)$ due to the initial prompt mismatch ($C \neq C^*$). This persistent deviation ensures that for most steps, the probability of generating the next correct token is suppressed relative to the ideal scenario. We can model this as:

$$P(w_t^*|C, w_1^*, \dots, w_{t-1}^*) = (1-\epsilon_t) \cdot P(w_t^*|C^*, w_1^*, \dots, w_{t-1}^*) \quad (6)$$

where $\epsilon_t \geq 0$ represents the per-step probability reduction due to the imperfect context. Substituting this into Equation 5 yields:

$$P(S^*|C) = P(S^*|C^*) \prod_{t=1}^{L} (1 - \epsilon_t) \quad (7)$$

Even if the per-step reductions ϵ_t are small, their cumulative product over a long sequence L causes the overall probability to diminish exponentially. As L increases, the term $\prod(1-\epsilon_t)$ rapidly approaches zero, leading to $P(S^*|C) \approx 0$.

This mathematical reality is exacerbated by the vastness of the token sequence space, $|\mathcal{V}|^L$. The set of syntactically and semantically valid programs S_{valid} constitutes an infinitesimally small manifold within this space. The compounding probability reduction described in Equation 7 ensures that any generative trajectory initiated from an imprecise prompt C is highly likely to deviate from this narrow manifold, making the generation of a specific target sequence S^* practically impossible.

F Data Construction Details

In this section, we provide a detailed breakdown of the construction pipeline for our training mixture, which consists of the task-specific **SRI-200K** dataset and the general **Glaive-Instruction** dataset.

F.1 SRI Dataset Construction

Source Data. We sourced raw code data from *The Stack v2* (Lozhkov et al., 2024), selecting a diverse set of repositories covering over 50 programming languages. To ensure data quality, we prioritized repositories with high star counts during the initial selection phase.

SRI Task Formatting. The core of our strategy lies in transforming static code into dynamic Search-and-Replace Infilling tasks. To simulate diverse real-world coding scenarios, we employed *tree-sitter* to parse the abstract syntax tree (AST) of the code and extract target "middle" segments based on structural granularity. We strictly enforced a task distribution ratio of **2:1:1:1** across four categories:

- **Function Body (2):** The model must complete the entire body of a function given its signature and surrounding context.
- **Multi-line Block (1):** The target is a logical block (e.g., `for` loops, `if-else` branches) identified via AST parsing.
- **Random Span (1):** An arbitrary span of code is removed to test robustness against unstructured boundaries.
- **Single Line (1):** Only a single line of code is masked, focusing on local syntax completion.

Sampling Strategy. From an initial pool of 200K generated samples, we sampled a high-quality subset of **20K samples** for fine-tuning. This selection was weighted based on repository star counts to prioritize code quality. Crucially, we avoided any content-based filtering on the "middle" (target) segments. This decision prevents the introduction of inductive bias, ensuring the model learns to generate code based on context and structure rather than memorizing specific "easy" patterns.

F.2 General Instruction Data

To maintain the model's chat and instruction-following capabilities, we utilized the *Glaive-Code-Assistant* dataset (Glaive AI, 2023). From this source, we sampled 60K pairs. We applied a heuristic filter to rigorously remove any samples related to "code completion," "infilling," or "search-and-replace" tasks. This strict separation ensures that the model learns the SRI format exclusively from our high-quality SRI-200K data, avoiding interference from lower-quality or differently formatted completion data.

F.3 Decontamination

To prevent data leakage, we performed strict decontamination against our evaluation benchmarks (CrossCodeEval, RepoEval, etc.). We identified and removed any training samples that originated from repositories present in the test sets. For benchmarks sourced from competitive programming (e.g., Codeforces), we ensured no overlap in problem IDs or descriptions.

G Design Decisions: Editable Region Sensitivity

To determine the optimal granularity for the Search-and-Replace Infilling task, we conducted a sensitivity analysis on the size of the editable region (target "middle" block). We trained a series of ablation models on SRI datasets constructed with varying target lengths: 5, 10, 15, 20, and 30 lines.

Table 10 summarizes the results across three representative benchmarks. We observe that performance remains highly stable for region sizes between 5 and 20 lines. However, a noticeable degradation occurs when the target expands to 30 lines (e.g., CrossCodeEval accuracy drops from 55.7 to 53.9).

Based on these empirical findings, we selected **10 lines** as the default configuration for our final

SRI-200K dataset. This choice strikes an optimal balance: it maintains peak code generation performance while offering superior token efficiency and a higher practical apply-rate compared to larger block sizes.

Configuration	CrossCodeEval	RepoEval	CrossCodeLongEval
SRI-5 lines	56.1	53.8	37.0
SRI-10 lines (Default)	**57.6**	**54.4**	**37.1**
SRI-15 lines	55.2	53.1	37.6
SRI-20 lines	55.7	53.4	36.9
SRI-30 lines	54.9	53.1	36.4

Table 10: Sensitivity analysis of the SRI editable region size. Performance (Exact Match) remains robust between 5 and 20 lines but degrades at 30 lines.

H Theoretical Analysis of Format Alignment

To elucidate the performance disparity between SRI and Natural Language (NL) FIM strategies observed in our main experiments, we hypothesize that **SRI's superiority stems from its structural alignment with the pre-training data distribution**, specifically the prevalence of Version Control System (VCS) diff patterns (e.g., Git commits).

H.1 Distributional Shift Hypothesis

Pre-trained Code LLMs are exposed to trillions of tokens, a significant portion of which are code repositories containing commit histories and diff files. These sequences naturally follow a `Context` $\rightarrow$ `Edit Pattern` (e.g., `«««< SEARCH`) structure. In contrast, NL-FIM imposes a conversational frame (e.g., "Please complete the code...") that introduces a **distributional shift**. While Chat models possess instruction-following capabilities, the transition from *conversational instruction* to *precise code insertion* is less represented in the high-quality code corpus compared to the *search-and-replace* pattern inherent in code evolution.

H.2 Experiment: Format Perplexity Analysis

To empirically validate this hypothesis, we designed a **Format Perplexity Analysis** to measure how "natural" each prompting format appears to the underlying model.

Setup. We randomly sampled 1,000 instances from the *CrossCodeEval* dataset. For each instance, we constructed inputs using three different prompting strategies: (1) Standard NL-FIM, (2) Dialogue NL-FIM, and (3) our SRI format.

Prompt for SRI

You are a code edit assistant. Your task is to implement ONLY the middle code that needs to be completed while keeping all other code exactly as is.
When you see a code file containing special comment markers /* MIDDLE CODE TO COMPLETE */, you should:

1. Generate a search/replace format output that:
- Identifies the exact region containing the /* MIDDLE CODE TO COMPLETE */ marker
- Provides the code that should replace the marker
2. Use the following format exactly:
“‘replace
«««< SEARCH
Code section containing /* MIDDLE CODE TO COMPLETE */ marker with enough context

=======
Same code section with ONLY the middle code implemented

»»»> REPLACE
“‘

3. Requirements:
- Only edit the code within a 10-line window around the identifier.
- The search section MUST contain the /* MIDDLE CODE TO COMPLETE */ marker

Figure 14: System prompts for our proposed SRI method. The SRI prompt explicitly instructs the model to generate a structured search-and-replace block.

Metric. We calculated the **Perplexity (PPL)** of the *ground truth middle code segment* conditioned on these different prompt prefixes using the **Qwen2.5-Coder-32B-Base** model. A lower PPL indicates that the model assigns a higher probability to the correct code under the given format, implying better alignment with its pre-training priors.

H.3 Results

The results, presented in Table 11, reveal a stark contrast. The SRI format achieves significantly lower perplexity compared to NL-based approaches.

Prompting Strategy	Format Type	Average PPL ↓
NL-FIM (Standard)	Conversational	5.42
NL-FIM (Dialogue)	Conversational	6.15
NL-FIM (Template)	Slot-filling	4.98
SRI (Ours)	**Diff / Edit**	**3.89**

Table 11: **Format Perplexity Analysis on Qwen2.5-Coder-Base.** The lower perplexity of SRI indicates that the diff-style format is significantly more aligned with the model's pre-training distribution than natural language instructions, enabling more confident and accurate code generation.

This empirical evidence supports our claim: **SRI effectively activates the latent "code editing" capabilities acquired during pre-training**, whereas NL-FIM forces the model to operate in a higher-entropy conversational mode that is suboptimal for precise code infilling.