Extending Fill-In-the-Middle with Instructions: Another "Free Lunch" for Code Completion Models

Anonymous ACL submission

Abstract

Code completion models often fail when the developer's intent is under-specified in the code context. To mitigate this, developers frequently use natural language comments to clarify objectives. However, current code completion models fail to prioritize these directives effectively since they are merely pre-trained using the Fill-In-the-Middle (FIM) objective. On the one hand, the natural language instructions, mixed with the noisy code comments, are just treated as part of the background context within the prefix. On the other hand, the pre-training datasets for the FIM objective are mostly sourced from open-source repositories, which results in a scarcity of high-intent instruction-to-code pairings that reflect the developers' workflow in code completion.

To bridge this gap, we propose Instruction-aware Fill-In-the-Middle (IFIM), a training method that extends the FIM structure with a dedicated instruction section. Our evaluation results indicate that IFIM significantly enhances model adherence to user intent, increasing the Pass@1 score from 84.6% to 93.6% without compromising underlying infilling performance. These findings suggest that IFIM offers a cost-effective "free lunch" for improving the steerability and practical utility of code completion systems. The artifacts are released at `https://anonymous.4open.science/r/ifim-artifacts`.

1 Introduction

Modern Integrated Development Environments (IDEs) rely on *code completion* to accelerate development (Wang et al., 2023): given a partial program, a model predicts the missing sections and displays them to the programmer. Large Language Models (LLMs) have pushed this feature from keyword-level suggestions to full-context generation, which adapts to different coding styles and languages with remarkable accuracy. As a result, coding assistants, with LLM-powered code completion as a key feature, have become indispensable tools in the developer community, such as Cursor, Github Copilot, and Trae.

Despite their prevalence, LLM-generated completions are not always helpful. Sun et al. (Sun et al., 2025) revealed that unaccepted completions often stem from *underspecified intent*, i.e., situations where the necessary context is absent from the code. For such context, it is almost impossible for LLMs, even for human experts, to infer the missing code. To resolve this ambiguity, developers frequently insert natural language instructions (e.g., inline comments like `# filter out negative numbers`) to clarify their goals. Indeed, studies indicate that programmers spend 11.56% of their time instructing these LLMs, comparable to the 14.05% spent on implementing functionality (Mozannar et al., 2024). Thus, the ability to accurately follow developer instructions is paramount for the utility of code completion tools.

Due to the unique nature of code completion tasks (i.e., the generated code must integrate correctly back into the original context), the code LLMs adopted for code completion are primarily pretrained models trained using the Fill-In-the-Middle (FIM) objective (Bavarian et al., 2022). However, FIM is not inherently designed to distinguish between background context and explicit developer intent. While it allows models to explicitly account for surrounding code, it treats comments containing natural language instructions as "background noise" within the prefix, indistinguishable from other unrelated comments. Consequently, models often fail to recognize these instructions as high-priority signals that should override or refine the completion logic. Furthermore, because most FIM pre-training data is sourced from open-source repositories where explicit "instruction-to-code" pairings are scarce, models lack exposure to the specific workflows used by developers to guide

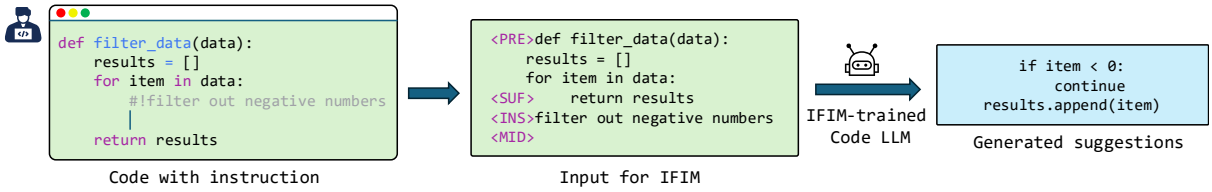

Figure 1: A usage example of IFIM-trained code LLMs. The code context is first processed into IFIM format and then infilled by the model. '#!' serves as a unique marker (configurable) that distinguishes an IFIM instruction from regular comments, enabling precise parsing.

code completion models.

To mitigate this, many AI IDEs deploy a secondary LLM specifically to edit code based on instructions. However, for tasks that a single code completion model could resolve given clear intent, this approach introduces unnecessary architectural complexity, prevents KV-cache reuse, and significantly increases latency. We argue that enhancing the instruction-following capability of the base code completion model is a more efficient and elegant solution.

A straight-forward solution is instruction-tuning (Wei et al., 2021): fine-tuning the LLM on (instruction, response) pairs so it learns to follow free-form natural language directives. Instruction-tuned code LLMs, such as Magicoder (Wei et al., 2023) and Wizardcoder (Luo et al., 2023), can achieve substantial improvements in code generation (i.e., generating a complete function from a natural language description). However, it backfires for FIM code completion. LLMs are found to exhibit a degradation in FIM code completion capabilities after the instruction-tuning training step (Pan et al., 2024; Ahmad et al., 2025; Wu et al., 2024), where an FIM-pretrained base model like CodeGemma-7b (Team et al., 2024) can easily outperform the much larger conversational LLM, GPT-4o, on real-world code completion benchmarks (Pan et al., 2024). This is because current instruction-tuning methods for code LLMs are designed for generating complete, assistant-style outputs. It leads to unpredictable behaviors, such as repeating the suffix or introducing unrelated code, in FIM code completion scenarios. Therefore, though the code LLMs' instruction-following capability can be enhanced, preserving the original FIM capability remains a challenge.

As revealed by OpenAI (Bavarian et al., 2022), the triple-section structure of FIM, i.e., prefix, suffix, and then middle, exhibits a "FIM-for-free" property, indicating that fill-in-the-middle can be learned without compromising left-to-right capabilities during pretraining. Inspired by this property, we investigate whether extending this structure can similarly preserve FIM capabilities while incorporating instructions. If so, the above-mentioned challenge could be resolved. Therefore, we propose Instruction-aware Fill-in-the-Middle (IFIM), which extends standard FIM by injecting an instruction section into the input sequence. IFIM is an off-the-shelf fine-tuning method applicable to existing FIM-trained code models. As shown in Figure 1, developers can specify instructions using a distinct marker, allowing the IFIM-trained model to explicitly follow their intent in practice.

To demonstrate the feasibility of IFIM, we constructed an synthesized training dataset using GPT-4.1 (gpt-4.1-2025-04-14) and fine-tuned two base code LLMs, DeepSeek-Coder (DeepSeek-AI et al., 2024) and Qwen2.5-Coder (Hui et al., 2024), with the IFIM objective. The trained models are evaluated on two code completion benchmarks IHumaneval and IRepoMasterEval, derived respectively from Humaneval-infilling (Fried et al., 2022) and RepoMasterEval (Wu et al., 2024) with human-annotated code completion instructions added to their data samples. Our experiments demonstrate that IFIM achieves remarkable improvements in instruction-following capability while maintaining the LLMs' ability to complete the code without instructions. Specifically, given the same instructions, our IFIM-trained variants can achieve better performance than the base model, with Pass@1 score averagely improved from 87.8% to 94.7% on IHumaneval and from 14.7% to 20.7% on IRepoMasterEval. Crucially, when no instructions are provided, IFIM models either match or slightly exceed their base counterparts' performance. Beyond clean instructions, IFIM also generalizes robustly to noisy developer comments (e.g., typos, verbose phrasing) and transfers across six programming languages despite training data being predom-

inantly Python. Moreover, IFIM introduces negligible inference overhead (0.04% additional input tokens), preserves suggestion diversity, and maintains clean edit locality, addressing the practical deployment concerns of production code completion systems. We also also conducted a series of ablation experiments to analyze the optimal position for inserting the instruction component, the ratio of IFIM processed data, the advantages of using a standalone instruction component and the alignment between instructions and generated completions. These results position IFIM as a practical, backward-compatible upgrade path for existing LLM-based code completion models.

Our contributions can be summarized as follows:

- We demonstrate that FIM can be further extended while maintaining its learning-for-free property.
- We propose IFIM, a simple yet effective finetuning method that helps FIM-pretrained models better follow natural language instructions without reducing their performance on non-instructed tasks.
- We conduct extensive experiments to show IFIM's dual benefits. Compared with base models, IFIM boosts average Pass@1 scores from 87.8% to 94.7% on IHumanEval and 14.7% to 20.7% on IRepoMasterEval while maintains performance when no instructions are given.

2 Related Work

2.1 Code LLMs for Code Completion

Code LLMs have become foundational in automated code completion. Early models like Codex (Chen et al., 2021), CodeGen (Nijkamp et al., 2022), and CodeGeex (Zheng et al., 2023) relied on left-to-right (L2R) generation, limiting their ability to handle real-world editing scenarios where code infilling is required. This gap led to the emergence of FIM training objectives (Bavarian et al., 2022; Fried et al., 2022), now adopted by state-of-the-art models including CodeGemma (Team et al., 2024), DeepSeek-Coder-V2 (DeepSeek-AI et al., 2024), and Qwen3-Coder (Yang et al., 2025). Various studies (Sagtani et al., 2025; Gong et al., 2025; Ding et al., 2024) are proposed to enhance FIM. For example, Sagtani et al. (Sagtani et al., 2025) introduce curriculum learning and context-aware sampling strategies to enhance FIM-based completion. Gong et al. (Gong et al., 2025) incorporate structural information (e.g., AST-guided masking) into the FIM pretraining process to better align the model with syntactic code editing. Ding et al. (Ding et al., 2024) propose a horizon-length prediction objective to dynamically guide the model's infilling scope. Despite these enhancements, both standard and improved FIM methods overlook the instruction-following capability of LLMs, a crucial requirement in real-world code completion scenarios (Mozannar et al., 2024). In this work, we fill this gap by proposing IFIM, a training framework that preserves the infilling capability of FIM while significantly improving the model's ability to follow instructions.

2.2 Instruction Tuning for Code LLMs

Instruction tuning, i.e., fine-tuning models on input-output pairs guided by natural language instructions, has proven to be a key technique for improving a model's ability to follow user intent. This approach has been widely applied to general-purpose conversational LLMs such as DeepSeek-R1 (Guo et al., 2025) and Qwen 3 (Yang et al., 2025). Instruction tuning has also shown promise in enhancing code LLMs for generation tasks. Models like Magicoder (Wei et al., 2023) and WizardCoder (Luo et al., 2023), which are tuned on relatively small instruction datasets, demonstrate strong performance on code generation benchmarks. However, multiple studies (Ahmad et al., 2025; Gong et al., 2024; Pan et al., 2024) reveal that instruction-tuned models remain suboptimal for code completion tasks, when compared to FIM-pretrained base models. These findings highlight a gap in unifying instruction-following and infilling capabilities. Inspired by the effectiveness of instruction tuning in natural language domains, we argue that an effective instruction-tuning method tailored for code completion is needed.

3 Instruction-aware Fill-In-the-Middle

3.1 Training Objective

IFIM further finetunes a FIM-pretrained code LLM, extending its capabilities to understand natural language instructions while preserving its foundational infilling capabilities. Each training code sample in the training dataset is the program partitioned into three segments, prefix (P), middle (M), suffix (S), with an instruction (I) describing the middle segment. During the finetuning, the model learns to predict the middle segment M con-

ditioned on the combined context of P, S, and I. This forms a supervised fine-tuning task where the objective is to maximize the log-likelihood of the target sequence:

$$\max_{\theta} E(P, M, S, I) \sim \mathcal{D}\left[\log P_{\theta}(M|P, S, I)\right]$$

where $\mathcal{D}$ is the training dataset, and the input is formatted according to a specific ordering of the components.

3.2 Training Sample Construction

Based on the training objective, we formalize the construction of input-output pairs for instruction tuning. The ground-truth output is the middle segment M. The input needs to be formatted following a specific mode, which specifies how the rest three components are represented. We term it as IFIM mode.

The base code LLMs to be tuned have their own FIM modes, where prefix, suffix, and middle are concatenated following a specific order with special tokens prepended to the beginning of each section. For example, a PSM mode indicates the order prefix, suffix, and middle. As an extension to FIM, IFIM preserves the relative ordering of the original FIM components (P and S) and insert the instruction I. For any given FIM pretraining mode (e.g., PSM), there are three potential IFIM modes (e.g., IPSM, PISM, PSIM). Let's use the PISM mode as an example. Its underlying FIM format (PSM) for input is:

$$\texttt{<PRE>} \circ P \circ \texttt{<SUF>} \circ S \circ \texttt{<MID>}$$

where $\circ$ denotes concatenation and <PRE>, <SUF>, and <MID> are the special tokens respectively leading the prefix, suffix, and middle sections. For IFIM, we insert the instruction I at the designated position, i.e, between the prefix P and the <SUF> token for PISM. Similarly, the instruction I is also paired with a leading special token <INS>. Therefore, the input of the training sample in PISM mode will be constructed as follows:

$$\texttt{<PRE>} \circ P \circ \texttt{<INS>} \circ I \circ \texttt{<SUF>} \circ S \circ \texttt{<MID>}$$

The resulted input-output pairs are thus used to train the base code model. As a new token, <INS> needs to be included in the vocabulary of the base LLM's tokenizer. However, adding the new tokens to the vocabulary would reinitialize the language model head, risking knowledge forgetting. To avoid this, we repurpose an existing but rare token from the vocabulary (e.g., the one with the lowest frequency) to serve as the <INS> token.

3.3 Usage for Code Completion

An IFIM-trained LLM enables a developer to guide code completion by embedding instructions directly within their code. Notably, IFIM does not require developers typing instructions for every single completion they want. Instead, it serves as an effective solution to mitigate the unhelpful code completions caused by the under-specified developer intent. Since plain text would violate the programming language's syntax, we propose writing instructions as specially marked inline comments. An example of this is shown in Figure 1. In this example, a developer using Python could write "#!filter out negative numbers" as the instruction. The '#!' prefix acts as a unique marker that distinguishes an IFIM instruction from a regular comment. This allows client-side tools, like IDE plugins, to build user-friendly features around it. For example, an IDE could be configured to automatically remove the instructional comment once the generated code is accepted by the user. When a code completion request is made, the code completion system parses the code around the cursor to extract the prefix and suffix, and identifies the specially marked comment as the instruction. These three components, prefix, suffix, and instruction, are then formatted into the required IFIM input sequence. The IFIM-trained LLM receives this input and generates the code to fill the middle, which is then presented to the developer as a completion suggestion.

4 Experiment Setup

This section presents the experimental configuration, including training dataset construction, baseline models, evaluation benchmarks, and implementation details.

4.1 Training Dataset

IFIM relies on training datasets consisting of code samples with code completion instructions. Ideally, the IFIM method can work with the production data collected by the real-world code completion applications, where such developer instructions for a specific code completion request can be easily obtained. Limited by our access to such data, we alternatively create a synthesized one as

a proxy to demonstrate its feasibility in our experiments. It is a well-established strategy that has demonstrated effectiveness in various studies (Wei et al., 2023; Luo et al., 2023; Huang et al., 2024). Specifically, we first source various code snippets from OSS-Instruct (Wei et al., 2023) and evol-codealpaca (Luo et al., 2023), two popular open-source instruction datasets. For each data point, we retain the code samples while removing the original instruction components, yielding 122.9k multi-language samples with 70.0% written in Python. For each code snippet, we randomly select 1-3 contiguous lines as the middle span and employ OpenAI GPT-4.1 (gpt-4.1-2025-04-14) as the instruction synthesizer to generate corresponding natural language descriptions for the middle span. The detailed prompts are available in Appendix A.

Moreover, we perform two data filtering steps on the resulted dataset. To ensure instructions are concise, we decontaminate the dataset by discarding any generated instructions longer than a single sentence. While some of the remaining instructions may have minor imperfections (e.g., being overly brief or vague), we retain them as they reflect realistic scenarios developers often face. Additionally, following (Huang et al., 2024), we exclude any samples that match problems from common evaluation benchmarks, such as HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021).

4.2 Base Code LLMs

We evaluate our approach using two popular code LLMs as base models: **Deepseek-Coder** (6.7B) (Guo et al., 2024) and **Qwen-2.5-Coder** (7B) (Hui et al., 2024). This model scale is comparable to those used in production code completion systems (e.g., JetBrains' Mellum (Semenkin, 2025) at 4B parameters), which prioritize low-latency inference (Wang et al., 2023). Each model family includes a base version (pretrained with FIM objectives) and an instruction-tuned version (further fine-tuned on proprietary instruction datasets). In our experiments, we apply IFIM training to the base versions since they are the LLMs used for code completion. For a comprehensive comparison, both the original base models and their official instruction-tuned counterparts serve as baselines in our evaluation.

4.3 Evaluation Benchmarks

We assess the instruction-following capabilities of our models using benchmarks derived from two complementary ones: IHumanEval from HumanEval-infilling and IRepoMasterEval (IRME) from RepoMasterEval, which respectively represent controlled and real-world evaluation scenarios.

- **HumanEval-infilling** (Fried et al., 2022) is a widely-used FIM benchmark constructed by removing middle spans from the canonical solutions in HumanEval. We adopt its single-line version, consisting of 1,640 infilling tasks. Each task includes a Python function with one random line of missing code, test cases, and docstrings describing the function's functionality. While widely adopted, this benchmark has notable limitations for real-world applicability: its function-level scope and well-documented examples differ from practical code completion scenarios. However, considering its popularity, we still adopt it as one evaluation benchmark for our experiments. To better simulate real-world conditions and increase the code completion difficulty, we remove the docstrings from all evaluation samples.

- **RepoMasterEval (RME)** (Wu et al., 2024) provides a more realistic evaluation framework for code completion models, which is used by ByteDance to evaluate their commercial code completion models. The pass rate on this benchmark is found to be highly correlated with the online user acceptance rate. RepoMasterEval is built upon real-world code repositories, spanning six languages: C++, Go, Java, JavaScript, Python, and TypeScript. It extracts the code files from various repositories and processes them into code infilling tasks, i.e., code files with missing middle spans. The model outputs are evaluated using the unit tests of the repository, enhanced through mutation testing and manual case additions. Due to computational constraints limiting our models' context windows, we retain the last 20 lines of the prefix and the first 20 lines of the suffix for each sample.

Notably, these two benchmarks are not paired with code completion instructions. To enable the evaluation, we synthesize instructions for each sample using OpenAI GPT-4.1 (prompts available in Appendix A) and then manually revise those unlike real-world instructions. Given the large sample

sizes, we randomly select statistically representative subsets from each benchmark (95% confidence level, 5% margin of error, 50% population proportion), resulting in 312 and 256 samples for HumanEval-infilling and RepoMasterEval, respectively. The manual revising process involves two authors independently reviewing all generated instructions and revising those that inadequately describe the desired code functionality. The authors achieve an inter-rater agreement of 0.72 and collaboratively resolve disagreements through discussion. The final curated instruction benchmarks are respectively named as **IHumanEval** and **IRepoMasterEval (IRME)**. Consent for their release has been obtained from the annotators.

4.4 Implementation Details

In our experiments, we use the Huggingface Transformers (Wolf et al., 2020) library with PyTorch to implement the models. The models are trained on a machine with 128 vCPUs, 200GB RAM, and two RTX A6000 GPUs (48GB RAM). For Deepseek-Coder, we adopt a batch size 2 with 128 accumulation steps and 1216 context length. For Qwen2.5-Coder, we adopt a batch size 1 with 256 accumulation steps to fit our GPU memory. These models are trained with an Adafactor optimizer (Shazeer and Stern, 2018) for 2 epochs. We set the initial learning rate at 5e-5 with 15 warmup steps and a linear scheduler. During inference, we use the greedy decoding strategy, and a maximum of 128 tokens are generated. For FIM mode selection, we adopt the recommended FIM mode, i.e., PSM, based on the default usage example in their official documents. All training samples are converted to the IFIM format (100% ratio). An ablation on alternative data mixing ratios is provided in Appendix C.

5 Results

5.1 Feasibility of IFIM

We assess two aspects to demonstrate the feasibility of IFIM: (1) whether IFIM can improve the instruction-following ability of LLMs, and (2) whether IFIM can maintain the code completion ability when no instruction is given. We fine-tune base LLMs (DeepSeek-coder and Qwen2.5-coder) using the IFIM training dataset and compare them against three baselines: (1) the original base models (**base**), (2) the official instruction-tuned versions (**instruct**), and (3) models fine-tuned on the raw, unprocessed training dataset (**finetuned**). To ensure a fair comparison, the finetuned baseline is exposed to the same instructions as the IFIM variants during training, but injected as an inline comment appended to the end of the prefix rather than as a structurally separate component. This design isolates the effect of IFIM's structural separation from the effect of merely exposing the model to instruction-bearing data.

Table 1: Performance comparison with IFIM and baselines. The setting *w/ ins.* provides each completion request with its corresponding instruction, appended as a comment to the prefix for base and finetuned models, while *w/o ins.* omits the instruction. We abbreviate DeepSeek-Coder as DS-Coder, Qwen2.5-Coder as Qwen-Coder, and IHumanEval as IHE.

Setting	Model	Version	Mode	IHE	IRME
w/ ins.	DS-Coder	base	PSM	84.6%	10.9%
		instruct	prompt	39.4%	2.1%
		finetuned	PSM	84.6%	10.5%
		IFIM	IPSM	**92.6%**	**19.9%**
			PISM	**93.6%**	**21.1%**
			PSM	**92.3%**	**19.1%**
	Qwen-Coder	base	PSM	91.0%	18.4%
		instruct	prompt	21.8%	3.6%
		finetuned	PSM	90.8%	0.4%
		IFIM	IPSM	78.2%	12.9%
			PISM	**95.2%**	**19.5%**
			PSM	**95.8%**	**20.3%**
w/o ins.	DS-Coder	base	PSM	68.6%	7.4%
		instruct	prompt	29.8%	0.6%
		finetuned	PSM	72.4%	7.0%
		IFIM	IPSM	**77.9%**	**12.9%**
			PISM	**76.6%**	**16.0%**
			PSM	**78.2%**	**12.1%**
	Qwen-Coder	base	PSM	76.0%	10.2%
		instruct	prompt	20.8%	1.9%
		finetuned	PSM	76.2%	0.8%
		IFIM	IPSM	60.3%	9.0%
			PISM	**76.3%**	**13.7%**
			PSM	**76.3%**	**13.3%**

In the "with instructions" setting, we pair each completion request with its corresponding instruction. For the base and finetuned models, instructions are written as a comment appended to the end of the prefix, mirroring the real-world practice where developers use inline comments to clarify intent. For the IFIM variants, instructions are constructed following the IFIM format. For the instruct variants, we embedded the instruction within

a custom prompt designed for the fill-in-the-middle task (available in Appendix B). In the "without instructions" setting, inputs for the base and IFIM models are prefixes and suffixes without modification. For the instruct variants, however, we use the same prompt that asks the model to perform the fill-in-the-middle task, but without providing the instruction. The Pass@1 score is computed to assess the quality of the completion.

As reported in Table 1, the IFIM variants outperform the base models and the finetuned baselines when instructions are provided, across both benchmarks. In the meantime, when no instructions are provided, with the notable exception of the IPSM mode for Qwen2.5-Coder, the IFIM variants still maintain a comparable, or even better, performance than the base models. It demonstrates that IFIM not only preserves but can also boost the infilling capability acquired during FIM pre-training. Moreover, the performance of instruct models aligns with the results in previous studies (Ahmad et al., 2025; Gong et al., 2024; Pan et al., 2024), i.e., existing instruct models impair their capabilities of fill-in-the-middle code completion.

5.2 Analysis and Ablation Studies

Impact of IFIM Modes. As shown in Table 1, the instruction-following ability acquired from IFIM training is sensitive to the position of the Instruction component. Specifically, with the notable exception of the IPSM mode for Qwen2.5-Coder, all other seven IFIM variants outperform their respective base models on both benchmarks when instructions are provided. Even without instructions, most variants maintain or enhance performance over the base model, confirming the broad effectiveness of IFIM. Comparing the specific configurations, the PISM and PSIM modes demonstrate comparable performance, with IPSM performs worst.

Based on these observations, we recommend PISM or PSIM as the default configurations for practitioners, since both consistently deliver substantial gains across model families and benchmarks. Among them, PSIM is preferable when the base model uses a PSM pretraining mode, as it preserves the KV cache for the suffix and thus minimizes inference overhead. We suggest practitioners avoid placing the instruction before the prefix (IPSM), which underperformed on Qwen2.5-Coder in our experiments.

Impact of Standalone Instruction Section. Compared to treating instructions as a separate, standalone section within the format, a simpler, more straightforward solution would be to train LLMs directly using instructions embedded as in-line comments. Therefore, we seek to investigate the impact of training on this newly introduced Instruction component, compared with simply learning instruction through inline comments. Specifically, we derive two training datasets: one processed using the IFIM format and the other by appending the instruction as an inline comment to the prefix. Using these two derived datasets, we train each base model (DeepSeek-Coder and Qwen2.5-Coder), obtaining the IFIM and comment version. During inference, the IFIM models are provided with instructed inputs, and the comment variants are provided with commented inputs (i.e., instructions are appended to the prefix as an inline comment). The results presented in Table 2 reveal that simply adding instructions as inline comments is an ineffective and often detrimental strategy. This confirms a hypothesis: LLMs can be confused when instructional text is mixed with code as comments, negatively impacting their primary code completion capabilities. Therefore, the structural separation of instructions from the code, as implemented in IFIM, is important for the model to properly disambiguate, understand, and benefit from the provided guidance.

Table 2: IFIM vs. Commented FIM: Evaluating the Standalone Instruction Component

Model	Version	Mode	IHumanEval	IRME
Deepseek-Coder	base	PMS	84.6%	10.9%
	comment	PMS	79.0%	4.3%
	IFIM	PIMS	**95.9%**	**21.1%**
Qwen2.5-Coder	base	PSM	89.4%	18.4%
	comment	PSM	88.5%	18.8%
	IFIM	PSIM	**96.6%**	**20.3%**

Robustness to Noisy Instructions. Since real-world developer comments are often imperfect, we further evaluate whether IFIM remains effective under noisy instructions. We inject three types of noise into the IHumanEval instructions: Misspelled (randomly altering one character per instruction), Lengthy (expanding the instruction with two unrelated background sentences), and Ambiguous (randomly removing one word). Across all three noise types and both model families, IFIM-trained variants consistently outperform their base counterparts—for example, on DeepSeek-Coder

Table 3: Evaluating Instruction Alignment on Reversed Instructions for DeepSeek-Coder

Version	Instruction Type	IHumanEval	
		Pass@1	Alignment
base	Standard	84.6%	54.4%
	Reversed	59.6%	
ifim	Standard	**93.6%**	**80.9%**
	Reversed	**56.7%**	

under the Misspelled setting, Pass@1 increases from 82.2% to 91.0%; on Qwen2.5-Coder under the Lengthy setting, from 85.2% to 92.2%. This indicates that the instruction-following capability acquired through IFIM generalizes beyond clean, well-formed instructions to the kind of noisy directives developers naturally produce. Full results are reported in Appendix E.

Cross-Lingual Generalization. Although our training data is predominantly Python (70.0%), the IRepoMasterEval benchmark spans six languages: C++, Go, Java, JavaScript, Python, and TypeScript. We analyze the per-language results and find that IFIM-trained models consistently outperform the base models on every language where the benchmark provides discriminative signal, i.e., C++, Go, Python, and TypeScript, when instructions are provided. For instance, on C++, IFIM-trained DeepSeek-Coder achieves 17.9% Pass@1 compared to the base model's 14.3%. This demonstrates that the instruction-following capability induced by IFIM transfers across languages with substantially different syntax, even when those languages are underrepresented in the training data. The Java and JavaScript slices yield 0% for both models due to the difficulty of those repo-level tasks and small sample sizes (N=33 and N=7), providing no signal to differentiate methods. Per-language results are reported in Appendix F.

Practical IDE Considerations. We further evaluate IFIM along three IDE-relevant dimensions: latency overhead, suggestion diversity, and edit locality. IFIM adds only 0.04% input-token overhead on IRepoMasterEval, matches the finetuned baseline's suggestion diversity (1.20 vs. 1.23 unique-and-correct completions per task), and produces zero cases of unwanted context repetition. Full methodology and results are in Appendix G.

Alignment between Instructions and Generated Code. Existing benchmarks (IHumanEval, IRME) feature instructions consistent with the code context, making it difficult to attribute IFIM's performance solely to instruction-following capability rather than context understanding. To decouple these factors, we employed GPT-4.1 to rewrite IHumanEval's instructions, creating deliberate conflicts between the instruction and the original code context. The prompt and examples are available in Figure 2. We evaluated DeepSeek-Coder (base and IFIM) on these conflicting samples using both Pass@1 and human annotation. Following a similar protocol to our benchmark creation, two authors independently classified each code completion based on whether the code implements the logic from the instruction. Notably, the annotators are required to only consider the alignment of the code's logic with the instruction's intent, irrespective of the code's functional correctness. The evaluators achieved an inter-rater agreement of 0.69, and all disagreements were resolved through discussion to reach a final consensus. As shown in Table 3, the base model frequently ignored the conflicting instructions in favor of the context. Conversely, IFIM demonstrated superior instruction alignment. This is evidenced by a sharper drop in Pass@1 on the original tests: by successfully following the modified instructions, IFIM naturally failed the original test cases, confirming its enhanced instruction-following capability. Moreover, we also performed a case study available in Appendix D, where we find that the IFIM-trained model tends to comply with instructions even though those instructions contradict the original functional context.

6 Conclusion

In this paper, we introduced IFIM, a training method that enhances FIM-trained models by extending their training objective to include an explicit instruction component. Our evaluation on popular code LLMs demonstrates that IFIM significantly boosts instruction-following accuracy without compromising the models' original FIM performance. Moreover, IFIM generalizes robustly to noisy instructions and across multiple programming languages, suggesting broad applicability. These results validate IFIM as a backward-compatible and effective solution for creating more helpful and responsive code completion systems.

Limitations

Evaluation Benchmarks A potential limitation lies in the quality and scale of the instructions used

for evaluation. Since both IHumanEval and IRepoMasterEval are derived by injecting AI-generated instructions into existing benchmarks, the resulting instruction style could deviate from real-world developer comments. We mitigated this through a rigorous two-author manual revision process, achieving an inter-rater agreement of 0.72,

and further validated robustness through the noise-injection experiments reported in Section 5.2 and Appendix E. Moreover, due to context-window constraints on the evaluated models, IRepoMasterEval samples are truncated to the last 20 prefix lines and the first 20 suffix lines per file. These design choices may underrepresent the long-context repository completion behaviors increasingly important in modern IDE settings, and we leave large-scale long-context evaluation to future work.

Scope of Control Baselines. To isolate the impact of our method, we utilized a controlled 0% IFIM data ratio to confirm that performance gains stem from instruction integration rather than mere exposure to additional code. However, our current study is scoped to this comparison, focusing on demonstrating the feasibility of this ideas. An important future work is to explore the interplay between IFIM and other alignment techniques.

Model Scale. Our experiments use code LLMs at the 6.7B–7B scale, chosen to reflect the practical deployment constraints of production code completion systems where low-latency inference is critical. Validating IFIM on larger architectures, where instruction-following capabilities may emerge differently, is an interesting direction for future work.

Ethic Consideration

The proposed method primarily aims to improve developer productivity, which several ethical implications may exist. As with any generative AI tool, improved code generation capabilities could theoretically be misused to lower the barrier for generating malicious code (e.g., malware or exploits) via natural language instructions. While our evaluation focused on benign functional correctness , the underlying mechanism of better intent adherence applies broadly. Future work should investigate safety alignment specifically for instruction-driven code infilling to prevent the generation of harmful content.

References

Wasi Uddin Ahmad, Somshubra Majumdar, and Boris Ginsburg. 2025. From output to evaluation: Does raw instruction-tuned code llms output suffice for fill-in-the-middle code generation? *arXiv preprint arXiv:2505.18789*.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program synthesis with large language models. *arXiv preprint arXiv: 2108.07732*.

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*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri 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. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv: 2107.03374*.

DeepSeek-AI, Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y. Wu, Yukun Li, Huazuo Gao, Shirong Ma, Wangding Zeng, Xiao Bi, Zihui Gu, Hanwei Xu, Damai Dai, Kai Dong, Liyue Zhang, Yishi Piao, and 21 others. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv: 2406.11931*.

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. *arXiv preprint arXiv:2410.03103*.

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

Linyuan Gong, Alvin Cheung, Mostafa Elhoushi, and Sida Wang. 2025. Structure-aware fill-in-the-middle pretraining for code. *arXiv preprint arXiv:2506.00204*.

Linyuan Gong, Sida Wang, Mostafa Elhoushi, and Alvin Cheung. 2024. Evaluation of llms on syntax-aware code fill-in-the-middle tasks. *arXiv preprint arXiv:2403.04814*.

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

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

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, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, and 5 others. 2024. Qwen2.5-coder technical report. *Preprint*, arXiv:2409.12186.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*.

Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. 2024. Reading between the lines: Modeling user behavior and costs in ai-assisted programming. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, pages 1–16.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*.

Zhenyu Pan, Rongyu Cao, Yongchang Cao, Yingwei Ma, Binhua Li, Fei Huang, Han Liu, and Yongbin Li. 2024. Codev-bench: How do llms understand developer-centric code completion? *arXiv preprint arXiv:2410.01353*.

Hitesh Sagtani, Rishabh Mehrotra, and Beyang Liu. 2025. Improving fim code completions via context & curriculum based learning. In *Proceedings of the Eighteenth ACM International Conference on Web Search and Data Mining*, pages 801–810.

Anton Semenkin. 2025. Mellum: How we trained a model to excel in code completion. JetBrains AI Blog. Accessed: 2026-05-12.

Noam Shazeer and Mitchell Stern. 2018. Adafactor: Adaptive learning rates with sublinear memory cost. In *International Conference on Machine Learning*, pages 4596–4604. PMLR.

Zhensu Sun, Xiaoning Du, Fu Song, Shangwen Wang, Mingze Ni, Li Li, and David Lo. 2025. Don't complete it! preventing unhelpful code completion for productive and sustainable neural code completion systems. *ACM Transactions on Software Engineering and Methodology*, 34(1):1–22.

CodeGemma Team, Heri Zhao, Jeffrey Hui, Joshua Howland, Nam Nguyen, Siqi Zuo, Andrea Hu, Christopher A Choquette-Choo, Jingyue Shen, Joe Kelley, and 1 others. 2024. Codegemma: Open code models based on gemma. *arXiv preprint arXiv:2406.11409*.

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

Jason Wei, Maarten Bosma, Vincent Y Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. 2021. Finetuned language models are zero-shot learners. *arXiv preprint arXiv:2109.01652*.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Empowering code generation with oss-instruct. *arXiv preprint arXiv:2312.02120*.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, and 1 others. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*, pages 38–45.

Qinyun Wu, Chao Peng, Pengfei Gao, Ruida Hu, Haoyu Gan, Bo Jiang, Jinhe Tang, Zhiwen Deng, Zhanming Guan, Cuiyun Gao, and 1 others. 2024. Repomastereval: Evaluating code completion via real-world repositories. *arXiv preprint arXiv:2408.03519*.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

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

A Prompt for annotating the training and evaluation datasets

We present the entire code sample to the LLM, but wrap the middle part with <explain></explain> tags and request a concise, one-sentence functional description. The prompt is structured as follows:

System Prompt:
You are a senior software engineer. When given a code snippet containing sections marked with <explain></explain> tags, write a single, concise instruction that explicitly describes the functional purpose of the code to be implemented within the tagged area. Focus on what needs to be achieved (e.g., inputs, outputs, logic) without prescribing how to implement it (e.g., specific methods, libraries). Ensure clarity and brevity so a developer can directly translate the instruction into code. Only write the instruction, no other text.

User Prompt:
Explain the code in the <explain></explain> tags using one simple sentence:
```{language}
{code}
```

B Prompt for the instruct models in our experiments

The models are provided with the following prompt:

Prompt:
Below is a incomplete code implementation and the description of its missing part.
* Incomplete Code:
```{language}
{code}
```
* Description: {instruction}
Please write code to fill the [MASK] (multiple lines of code may be masked out). Your code should only include the code to fill the [MASK]. Do not repeat the context of the [MASK].

C Impact of IFIM Training Data Ratio

A natural design question for IFIM is how much of the training data should be converted to the IFIM format, as opposed to retained in the standard FIM format native to the base model. To investigate this, we experimented with five different data mixtures, converting 0%, 25%, 50%, 75%, and 100% of the training samples into the IFIM format, with the remainder kept in the standard FIM format. For this experiment, we used the best-performing IFIM mode for each model: PISM for DeepSeek-Coder and PSIM for Qwen2.5-Coder. The results are shown in Figure 3.

Overall, a full conversion of the training data to the IFIM format (100%) proves to be an effective strategy across both model families and benchmarks. However, the relationship between data ratio and model performance is not strictly linear and reveals complex trade-offs. The most striking example is Qwen2.5-Coder, where the 50% ratio causes a notable performance drop on IHumanEval (73.4%) compared to both lower and higher ratios, suggesting that the optimal mixture is model-dependent. We therefore recommend practitioners default to a 100% IFIM conversion, which consistently delivers strong results without requiring model-specific tuning of the data mixture.

D Case Study for Alignment

We further qualitatively demonstrate the instruction-following behaviors between base and IFIM models using five real examples, shown in Figure 2. In each case, the base model tends to ignore the commented instruction, defaulting to the logic suggested by the original function context. For instance, in Example 5 (HumanEval/63/L4), given the conflicting instruction that requires to check if n equals to 3, the base model still generates 'if n == 2:', the code expected by the code context. Conversely, the IFIM model adheres to the new instruction, producing 'if n == 3:'. These examples provide qualitative evidence that IFIM training genuinely enhances the model's ability to prioritize and follow explicit natural language instructions, even when they contradict the surrounding code context.

E Robustness to Noisy Instructions

Real-world developer comments are often imperfect: they may contain typos, be unnecessarily verbose, or omit relevant details. To assess whether IFIM remains effective under such conditions, we construct three noisy variants of the IHumanEval instructions:

- **Misspelled**: randomly alter one character in

Figure 2: A visual comparison of outputs produced by base and IFIM DeepSeek-Coder given conflicting instructions. The checkmark (✓) indicates that the generated code aligns with the intent of the Conflicting Instruction, while the cross mark (✗) indicates that it does not.

Example 1: HumanEval/143/L3
Original Instruction: Check if the length of the word is 1.
Conflicting Instruction: Check if the word starts with 'a'.

Base model: `if word[0] != 'a':` ✗
IFIM model: `if word[0] == 'a':` ✓

Example 2: HumanEval/140/L12
Original Instruction: Append the current non-space character.
Conflicting Instruction: Append an underscore character.

Base model: `new_text += text[i]` ✗
IFIM model: `new_text += "_"` ✓

Example 3: HumanEval/3/L4
Original Instruction: Check if the balance is less than zero.
Conflicting Instruction: Check if the balance is greater than zero.

Base model: `if balance < 0:` ✗
IFIM model: `if balance > 0:` ✓

Example 4: HumanEval/13/L0
Original Instruction: Repeat the loop as long as b is not zero.
Conflicting Instruction: Repeat the loop as long as a is not zero.

Base model: `while b != 0:` ✗
IFIM model: `while a != 0:` ✓

Example 5: HumanEval/63/L4
Original Instruction: Check if n equals to 2.
Conflicting Instruction: Check if n equals to 3.

Base model: `if n == 2:` ✗
IFIM model: `if n == 3:` ✓

each instruction, simulating typographical errors.

- **Lengthy**: expand each instruction with two unrelated background sentences, simulating verbose comments that mix intent with extraneous context.
- **Ambiguous**: randomly remove one word from each instruction, simulating the kind of elision common in informal developer notes.

Both DeepSeek-Coder and Qwen2.5-Coder are evaluated in their base and IFIM-trained variants. The IFIM variants use the best-performing mode for each model family (PISM for DeepSeek-Coder, PSIM for Qwen2.5-Coder). Results are shown in Table 4.

Across all three noise types and both model families, IFIM-trained variants consistently outperform their base counterparts. The improvement is largest under the Lengthy setting (e.g., +8.7 points on DeepSeek-Coder, +7.0 points on Qwen2.5-Coder), suggesting that the structural separation of instructions in IFIM helps the model isolate intent from surrounding noise. Even under the most degrading Ambiguous setting, where one word is randomly removed, IFIM retains a consistent advantage of 6–7 points over the base models. These results indicate that the instruction-following capability acquired through IFIM generalizes beyond clean, well-formed instructions to the kind of imperfect directives developers naturally produce in practice.

F Per-Language Results on IRepoMasterEval

IRepoMasterEval spans six languages: C++, Go,

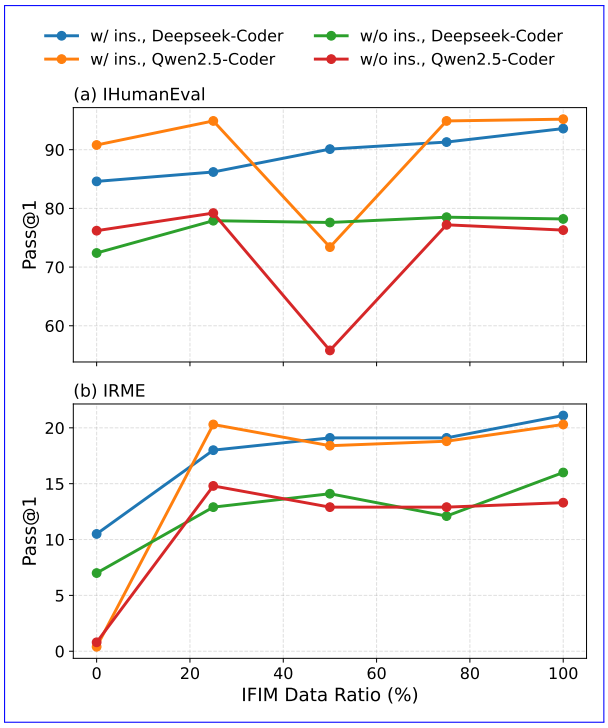

Figure 3: Impact of IFIM Training Data Ratio in Training on Model Performance

Table 4: Pass@1 on IHumanEval under three noise injection settings. **Bold** indicates IFIM outperforms its base counterpart.

Model	Setting	Version	Pass@1
DeepSeek-Coder	Misspelled	base	82.2%
		IFIM	**91.0%**
	Lengthy	base	82.9%
		IFIM	**91.6%**
	Ambiguous	base	77.4%
		IFIM	**84.5%**
Qwen2.5-Coder	Misspelled	base	89.9%
		IFIM	**93.3%**
	Lengthy	base	85.2%
		IFIM	**92.2%**
	Ambiguous	base	80.0%
		IFIM	**86.3%**

Java, JavaScript, Python, and TypeScript. Table 5 reports per-language Pass@1 for DeepSeek-Coder (base vs. IFIM-trained, PISM mode) under both "with instructions" and "without instructions" settings.

On the four languages with discriminative signal (C++, Go, Python, TypeScript), IFIM consistently outperforms the base model when instructions are provided. The instruction-following capability transfers across all four languages despite the training data being dominated by Python (70.0%). When no instructions are provided, IFIM maintains or improves performance on three of the four languages, with Python showing a modest decrease (from 23.7% to 21.1%) that is offset by substantial gains in the instruction-conditioned setting.

G Practical IDE Considerations

Beyond Pass@1, deploying IFIM in a real IDE raises three practical concerns: latency overhead, suggestion diversity, and edit locality. We report the methodology and detailed results for each below.

Latency Overhead. A code completion model must respond within tight latency budgets to remain usable. IFIM introduces exactly one additional special token (<INS>) per request beyond the natural-language tokens of the instruction itself. To quantify the structural overhead introduced by IFIM (independent of instruction length, which is under the developer's control), we measure the proportion of total input tokens contributed by this single special token across all IRepoMasterEval samples. The overhead averages 0.04% of the total input tokens, making the structural cost of IFIM negligible relative to typical code-completion latency budgets. The instruction text itself contributes additional tokens proportional to the instruction length, but this is identical to the cost of writing the comment in the base setting.

Suggestion Diversity. Code completion systems often present multiple candidate completions to the developer. A common concern with fine-tuned models is that they may collapse to a narrow output distribution, reducing the utility of multi-suggestion interfaces. To assess whether IFIM preserves output diversity, we sample five completions per task on IHumanEval using DeepSeek-Coder with temperature $t = 0.2$ and count the number of *unique-and-correct* suggestions per task (i.e., suggestions that pass the unit tests and are textually distinct from one another). IFIM produces an average of 1.20 such suggestions per task, closely matching the finetuned baseline's 1.23. This indicates that IFIM does not meaningfully reduce suggestion diversity compared to standard fine-tuning.

Edit Locality. A common failure mode of standard instruction-tuned models in code completion settings is re-emitting tokens already present in the

Table 5: Per-language Pass@1 on IRepoMasterEval (DeepSeek-Coder, PISM mode). **Bold** indicates IFIM outperforms its base counterpart on that language.

Setting	Version	Language	N	Pass@1
Inputs with instructions	base	C++	28	14.3%
		Go	109	5.5%
		Java	33	0.0%
		JavaScript	7	0.0%
		Python	38	34.2%
		TypeScript	50	10.0%
	IFIM	C++	28	**17.9%**
		Go	109	**12.8%**
		Java	33	0.0%
		JavaScript	7	0.0%
		Python	38	**50.0%**
		TypeScript	50	**22.0%**
Inputs without instructions	base	C++	28	10.7%
		Go	109	3.7%
		Java	33	0.0%
		JavaScript	7	0.0%
		Python	38	**23.7%**
		TypeScript	50	6.0%
	IFIM	C++	28	**14.3%**
		Go	109	**11.9%**
		Java	33	0.0%
		JavaScript	7	0.0%
		Python	38	21.1%
		TypeScript	50	**20.0%**

surrounding context. For example, repeating the leading tokens of the suffix or trailing tokens of the prefix within the generated middle segment. Such redundancy breaks edit locality and forces the IDE to perform post-hoc deduplication. We check for this failure mode programmatically by scanning each IFIM-generated completion for overlap with the leading tokens of the provided suffix or the trailing tokens of the prefix. Across both IHumanEval and IRepoMasterEval, our script initially flagged 4.2% of IFIM-generated samples as containing potential overlap. Subsequent manual inspection of all flagged cases confirmed **zero** actual instances of unwanted context repetition. This indicates that IFIM avoids the boundary-blurring behaviors common to standard instruction-tuned variants, preserving the clean edit locality expected of FIM-trained models.

H Marker Parsing in IDEs

IFIM relies on a designated marker (e.g., #! in Python) to distinguish instruction comments from ordinary comments. The parsing logic is minimal: it requires only that the IDE's lexer recognize comments and check for the predefined marker prefix, a capability supported by all modern IDEs. The marker is configurable per language and can be made more complex (e.g., #!ai:) to reduce the risk of accidental matches with naturally occurring comments in the wild.