Breaking Memorization Barriers in LLM Code Fine-Tuning via Information Bottleneck for Improved Generalization

Anonymous ACL submission

Abstract

Adapting pretrained large language models (LLMs) to code domains via supervised fine-tuning (FT) has been commonly used for code generation. However, we identify a previously underappreciated failure mode, the *memorization barrier*, where strong memorization of downstream code data in the base model could trap optimization and prevent the standard FT from effectively acquiring new, generalizable code knowledge. To overcome this barrier, we propose the *information bottleneck (IB)-guided fine-tuning*, termed IB-FT, which applies an IB penalty on hidden representations of the code data to compress spurious, memorized features while preserving task-relevant information. Extensive experiments on two code benchmarks (OriGen and Evol-CodeAlpaca-V1) show that IB-FT substantially alleviates the memorization barrier, improves top-1 performance (Pass@1), and yields far more stable gains under the stricter multi-sample metric Pass@$k^{(m)}$ (a problem counts as solved only if at least m of k samples pass unit tests) compared with conventional FT.

1 Introduction

Code generation sits at the nexus of AI and software engineering, driving rapid advances in both research and industry (Nijkamp et al., 2023; Hou et al., 2024; Thakur et al., 2024; Huynh and Lin, 2025; Li et al., 2022). By automating programming tasks, LLMs can boost developer productivity, lower engineering costs, and increase accessibility, contributions already translate to billions in annual economic value and are expected to grow as adoption widens (Daniotti et al., 2025).

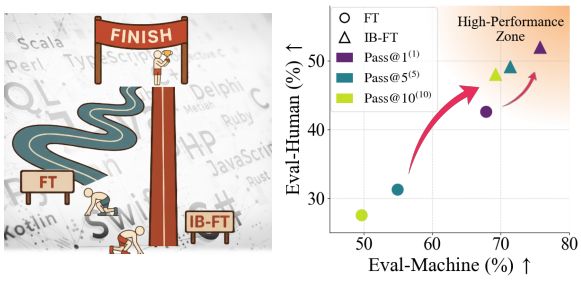
Yet, rather than training models from scratch on specialized code corpora, practitioners typically fine-tune pretrained LLMs on downstream code datasets because this approach is computationally cheaper and more accessible (Roziere et al., 2023; Beckmann et al., 2004; Wang et al., 2022). Nevertheless, adaptation remains challenging: models must acquire both programming syntax and compositional semantics while avoiding overfitting to dataset idiosyncrasies and excessive memorization of pretraining data (Mathews and Nagappan, 2024; Xu et al., 2022; Fakhoury et al., 2024). In this work, we show that code fine-tuning is often brittle: fine-tuned models can exhibit a large gap between greedy decoding (Pass@1) and sampling-based metrics (Pass@k), *i.e.*, a correct program may appear among multiple (k) sampled generations while the one-time generation by Pass@1 remains incorrect; We defer further details to Fig. 2 (Sec. 3). This evidence exposes the limited effectiveness of current code fine-tuning practices and motivates our central question:

> ***(Q)*** *What causes the ineffectiveness of LLM code fine-tuning, and how can fine-tuning be modified to reliably improve generalization?*

To tackle **(Q)**, we first investigate why code fine-tuning is not effective to acquire new code knowledge, examining the interaction between fine-tuning data and the pretrained model through the lens of *memorization*. We identify a "***memorization barrier***": the base model already strongly memorizes the fine-tuning code data, trapping optimization in a region that the standard fine-tuning objective cannot escape. Although prior work has identified memorization as a core challenge for LLM-based code generation (Chen et al., 2025; Yang et al., 2024; Al-Kaswan et al., 2024), most studies treat memorization as a privacy or contamination problem, either by detecting leakage against pretraining or fine-tuning corpora (Carlini et al., 2022; Zeng et al., 2023) or by showing that test-set contamination in pretraining inflates evaluation scores (Deng et al., 2024; Dong et al., 2024; Golchin and Surdeanu, 2025; Wang et al., 2025; Riddell et al., 2024). In contrast, we adopt a cross-

setting perspective and show that strong memorization of the fine-tuning dataset already present in the base model (prior to adaptation) creates a "memorization barrier"[1] that traps optimization and prevents effective, generalizable fine-tuning.

To overcome the memorization barrier, we also propose a novel fine-tuning (FT) strategy, termed *IB-regularized fine-tuning* (IB-FT), motivated by the information bottleneck (IB) principle (Tishby and Zaslavsky, 2015; Saxe et al., 2019). IB-FT applies an IB regularizer to hidden representations to compress prediction-irrelevant (spurious) features that arise from memorization bias. By constraining representation capacity and reshaping the data representation distribution, IB-FT reduces the dominance of heavily memorized samples and promotes more uniform learning across the code dataset, improving domain adaptation and generalization to unseen code. Intuitively, consider two runners: one starts early but follows a winding, indirect path (standard FT, which appears ahead by memorization yet is slowed by detours), while the other starts at the official line but runs straight to the finish (IB-FT, which, despite a later start, reaches the goal more directly). See **Fig. 1** for an illustrative schematic and performance comparison.



(a) Code fine-tuning race (b) Fine-tuning on OriGen

Figure 1: Schematic overview of IB-FT versus conventional FT and a representative performance comparison when fine-tuning on OriGen. (a) A code fine-tuning "race": conventional FT (left) appears to advance quickly by memorization but follows a winding, less-generalizable path, whereas IB-FT (right) progresses along a straighter trajectory toward the "winning" generalization. (b) Test-time performance for DeepSeek-Coder-7B-Instruct-v1.5, reported on Eval-Human and Eval-Machine using Pass@$k^{(m)}$ with $m = k \in \{1, 5, 10\}$ (a problem is counted only if all k samples pass). IB-FT consistently attains higher, more stable accuracy and occupies the high-performance region compared to FT.

We summarize our main contributions below:

- We introduce and formalize the memorization barrier problem, showing via systematic analysis that strong pre-existing memorization of fine-tuning examples can critically limit the effectiveness of code fine-tuning.
- We recast fine-tuning through the IB (information bottleneck) lens and propose an IB-based regularizer (IB-FT) that compresses spurious features, alleviates the memorization barrier, and promotes more effective code fine-tuning.
- We present extensive empirical results validating the memorization barrier phenomenon and benchmark IB-FT on two diverse code datasets, OriGen(Cui et al., 2024) and Evol-CodeAlpaca-V1 (Luo et al., 2024), showing that IB-FT consistently improves generalization, including under stricter evaluations such as Pass@1.

2 Related Work

LLM code fine-tuning. Prior work on code fine-tuning has improved model performance via data-side interventions, *e.g.*, synthetic or augmented corpora (Luo et al., 2023; Li et al., 2023), and method-side advances, *e.g.*, multi-task FT, self-alignment, and parameter-efficient tuning (Ma et al., 2023; Wei et al., 2024; Liu et al., 2024a; Zhuo et al., 2024). However, these efforts largely ignore memorization in the training data, preventing the potential of code datasets from being fully exploited.

Information bottleneck (IB) in LLMs. The information bottleneck (IB) principle (Tishby and Zaslavsky, 2015; Shwartz-Ziv and Tishby, 2017; Yang et al., 2025) prescribes that latent representations should retain task-relevant information while discarding irrelevant details. Prior IB-based work has applied this idea to reasoning, calibration, security, and representation analysis, emphasizing compression or mutual-information estimation (Chen et al., 2024; Li et al., 2025; Liu et al., 2024b; Lei et al., 2025). By contrast, to the best of our knowledge, we are the first to apply IB-guided fine-tuning in code generation, specifically to suppress spurious memorized features during adaptation.

3 Preliminaries and Challenges in LLM Code Fine-tuning

Preliminaries on LLM code fine-tuning. LLMs are pretrained on massive and heterogeneous corpora, which equip them with broad linguistic and reasoning capabilities. While such pretraining confers strong general abilities, it often falls short in providing the specialized expertise required for code-related tasks, *e.g. reasoning beyond code syn-*

[1] Although introduced for code generation, the memorization barrier likely extends to other domains where pretrained models already memorize downstream data, thereby hindering effective adaptation.

tax (Jain et al., 2023), Python code generation (Le et al., 2022), domain-specific adaptation for Verilog code generation (Thakur et al., 2024; Zhao et al., 2024). Instead of training an LLM from scratch on code-centric corpora, the conventional approach is to adapt a pretrained model through *code fine-tuning*, where specialized code datasets are used to align the model with the target code distribution (Cui et al., 2024; Wei et al., 2025).

Formally, given a pretrained model with parameters $\boldsymbol{\theta}_0$ and a target-domain code dataset $\mathcal{D}_{\text{code}} = \{(x, y)\}$, where x denotes the input (*e.g.*, problem description) and y denotes the target output (*e.g.*, solution code), the fine-tuning objective is to minimize the negative log-likelihood:

$$\underset{\boldsymbol{\theta}}{\text{minimize}}\ \mathbb{E}_{(x,y)\sim\mathcal{D}_{\text{code}}}\big[-\log p_{\boldsymbol{\theta}}(y \mid x)\big], \tag{1}$$

where $\boldsymbol{\theta}$ are the model parameters initialized from $\boldsymbol{\theta}_0$. LLM-based code models drive applications such as code completion, program synthesis, and translation. Their potential to enhance productivity, reduce errors, and democratize programming underscores the need for effective code fine-tuning.

As for Verilog code generation evaluation, *VerilogEval* is commonly used benchmark (Liu et al., 2023), which consists of two complementary subsets. *Eval-Machine* refers to automatic evaluation, where generated code is executed against unit tests or reference outputs (Liu et al., 2023). *Eval-Human*, in contrast, relies on human annotators to assess the quality of generated code in terms of correctness, clarity, and style (Liu et al., 2023). The benchmark uniformly adopts the well-known Pass@k metric, which measures the probability that at least one of the top-k generated samples passes the unit tests (Chen et al., 2021).

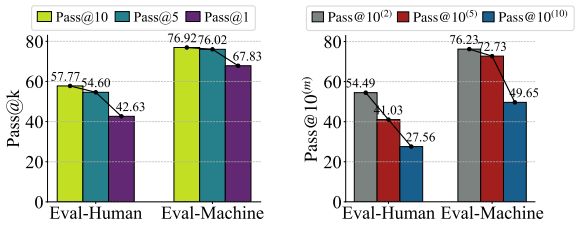


Figure 2: Performance of DeepSeek-Coder-7B-Instruct-v1.5 fine-tuned on the OriGen dataset, evaluated on the Eval-Human and Eval-Machine. (a) Pass@k results for $k \in \{10, 5, 1\}$, measuring the probability that at least one of the top-k generations passes the test cases. (b) Pass@$k^{(m)}$ results with $k = 10$ and $m \in \{2, 5, 10\}$, requiring multiple successful generations among the ten samples to reflect robustness.

An illusion of effectiveness in code fine-tuning. Before our study, it was commonly believed that code fine-tuning delivers acceptable performance, particularly under the *Pass@k* metric, which measures the probability that *at least one* of the top-k generated outputs from an LLM is correct. However, we find that this might be an illusion of effectiveness: (1) under greedy decoding (*i.e.*, $k = 1$), the fine-tuned LLM can experience a sharp accuracy drop, and (2) even for larger k, most generations actually fail despite one occasionally passing.

Fig. 2 validates the above two limitations by fine-tuning the pretrained DeepSeek-Coder-7B-Instruct-v1.5 on the OriGen, with test-time generalization performance evaluated on Eval-Human and Eval-Machine. As shown in Fig. 2(a), the fine-tuned model performance, evaluated by both *Eval-Machine* and *Eval-Human*, exhibits a significant accuracy drop at Pass@1. This indicates that the fine-tuned model lacks certainty under greedy decoding, unless one leverages the strength of probabilistic generation. Furthermore, to probe beyond the standard Pass@k, we introduce a complementary metric *Pass@$k^{(m)}$*, which measures the probability that *at least* m of the top-k generated outputs are correct. Note that Pass@$k^{(1)}$ reduces to the classic Pass@k evaluation. As shown in Fig. 2(b), the accuracy under Pass@$k^{(m)}$ decreases sharply as m increases, revealing that even when allowing multiple generations (*e.g.*, $k = 10$), most outputs are still incorrect; for instance, $m = 5$ requires half of the generations to succeed, yet performance drops drastically in this setting.

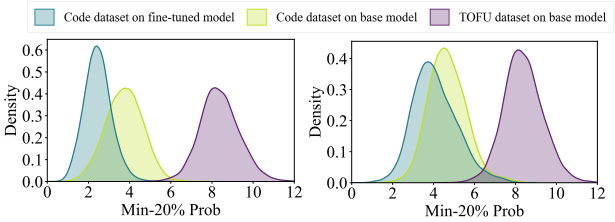
Problem statement. As shown in Fig. 2, achieving effective code fine-tuning for LLMs is far from trivial, with root causes largely overlooked in prior work. To address this gap, we focus on two key questions in this study: ***(Q1)*** *What causes the difficulty of effective code fine-tuning for LLMs?* ***(Q2)*** *How can the existing fine-tuning protocol in (1) be advanced to achieve improved effectiveness?*

In the rest of the paper, we address (Q1) by identifying the "memorization barrier" in LLM code fine-tuning, where excessive memorization of fine-tuning data by the pretrained LLM can *hinder* effective adaptation (see Sec. 4). We then address (Q2) by tackling this barrier through an information bottleneck perspective, enforcing *equality* across data with different memorization levels so that the fine-tuner learns from the most informative aspects of the data (see Sec. 5).

4 The "Memorization Barrier" in LLM Code Fine-tuning

The memorization lens on (in)effective code fine-tuning. As illustrated in Fig. 2, the code fine-tuned LLM does not appear to gain much from the code dataset, as evidenced by its sharp accuracy drop in Pass@1 and in Pass@$k^{(m)}$ for large m. This raises the question of whether the limitation stems from the quality of the data themselves or from the fine-tuning process failing to fully exploit the code data. In this work, we assume no control over data curation and instead focus on improving the fine-tuning process.

Our hypothesis is that effective code fine-tuning should enable the model to acquire new domain knowledge beyond pretraining, which motivates a deeper examination of the relationship between fine-tuning data and the pretrained LLM. We approach this from the perspective of *memorization* (Shi et al., 2023; Biderman et al., 2023; Leybzon and Kervadec, 2024; Kiyomaru et al., 2024): Given that the pretraining corpus is massive and may partially overlap with the target domain, we examine whether a non-trivial portion of the fine-tuning data has already been memorized, and how this entanglement, ignored in the conventional fine-tuning protocol 1, underlie the observed limitations.



(a) OriGen (b) Evol-CodeAlpaca-V1

Figure 3: Downstream data already memorized during pretraining. Memorization patterns measured by `Min-K% Prob` ($K = 20$) using DeepSeek-Coder-7b-Instruct-v1.5 and Llama-3 8B as base models on two code datasets, OriGen and Evol-CodeAlpaca-V1. The x-axis shows `Min-20% Prob` scores and the y-axis their density. For each dataset, three distributions are reported: $p_0(\mathcal{D}_{\text{code}}; \boldsymbol{\theta}_0)$ (code dataset on the base model), $p_1(\mathcal{D}_{\text{code}}; \boldsymbol{\theta}_{\text{code}})$ (code dataset on the fine-tuned model), and $p_2(\mathcal{D}_{\text{TOFU}}; \boldsymbol{\theta}_0)$ (TOFU dataset on the base model), where p_2 provides the weakest-memorization reference.

Through the lens of memorization, **Fig. 3** shows the memorization scores of the fine-tuning code datasets OriGen (Cui et al., 2024) and Evol-CodeAlpaca-V1 (Luo et al., 2023) with respect to the base model ($\boldsymbol{\theta}_0$). Here, memorization is assessed using the `Min-K% Prob` method (Shi et al., 2023), which measures how likely a sequence has been memorized by a model. It selects the $K\%$ tokens with the lowest predicted probabilities and computes their average negative log-likelihood (*i.e.*, the prediction loss over the least likely tokens). A lower `Min-K% Prob` score indicates stronger memorization, as even the least likely tokens are predicted confidently, whereas a higher score suggests weaker memorization.

In Fig. 3, we present the distribution of `Min-K% Prob` (with $K = 20$) scores for the fine-tuning code dataset (OriGen or Evol-CodeAlpaca-V1, denoted $\mathcal{D}_{\text{code}}$) on the pretrained model ($\boldsymbol{\theta}_0$) and its fine-tuned counterpart ($\boldsymbol{\theta}_{\text{code}}$), denoted as $p_0(\mathcal{D}_{\text{code}}; \boldsymbol{\theta}_0)$ and $p_1(\mathcal{D}_{\text{code}}; \boldsymbol{\theta}_{\text{code}})$, respectively. For comparison, we also include $p_2(\mathcal{D}_{\text{TOFU}}; \boldsymbol{\theta}_0)$, the distribution obtained from the base model evaluated on the fictitious TOFU dataset (Maini et al., 2024), which is excluded from pretraining as it contains synthetic author profiles. The rationale is that $p_2(\mathcal{D}_{\text{TOFU}}; \boldsymbol{\theta}_0)$ serves as *a reference distribution for the weakest memorization* of the base model, since $\mathcal{D}_{\text{TOFU}}$ contains entirely fictitious information. As shown in Fig. 3, p_0 lies to the right of p_1, reflecting the stronger memorization of $\mathcal{D}_{\text{code}}$ by the fine-tuned model $\boldsymbol{\theta}_{\text{code}}$. This is expected, since $\boldsymbol{\theta}_{\text{code}}$ is directly trained on $\mathcal{D}_{\text{code}}$. More interestingly, *however*, p_0 has shown substantial overlap with p_1, suggesting that *the base model has already memorized much of the dataset prior to fine-tuning*. This strong pre-existing memorization becomes especially evident when contrasted with the weak memorization reference p_2.

The above reveals a *new memorization phenomenon* identified from LLM code fine-tuning: fine-tuning data points may already be strongly memorized by the pretrained base model even before fine-tuning. We remark that this finding differs from prior literature, which has typically examined memorization from a privacy perspective (to detect leakage), either by evaluating an LLM against its pretraining data (Carlini et al., 2022) or by assessing a fine-tuned LLM against its fine-tuning data (Zeng et al., 2023). In contrast, ours introduces a *cross-setting* perspective: evaluating the memorization of *fine-tuning datasets* on the *base model*.

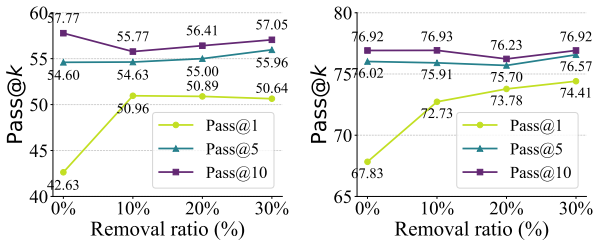
"Memorization barrier" in code fine-tuning. Given the strong memorization of code fine-tuning data by the pretrained base model, we posit that this memorization underlies the limited effectiveness of LLM code fine-tuning. This challenge is analogous to a known challenge in nonconvex optimization, "*escaping bad local optima*" (Ge et al., 2015; Criscitiello and Boumal, 2019). Our rationale is that,

due to the memorization, the base model becomes trapped in a bad "local optimum" characterized by high memorization. Consequently, the conventional fine-tuning objective (1) struggles to escape this state and converge to a better solution, *i.e.*, a code fine-tuned model with improved generalization. We refer to this issue as "**memorization barrier**" and state it below.

> **Memorization barrier:** The phenomenon where LLM code fine-tuning starts from a base model $\boldsymbol{\theta}_0$ that already strongly memorizes the fine-tuning set $\mathcal{D}_{\text{code}}$, placing optimization in a state the conventional fine-tuning objective struggles to escape, thereby leading to poor generalization on downstream code tasks.

Fig. 4 validates the memorization barrier by comparing *Pass@k* performance of the fine-tuned model using the conventional approach (1) with different data pruning ratios, where the most memorized data points (identified in Fig. 3) are removed from the fine-tuning set. We examine whether excluding highly memorized fine-tuning data, *i.e.*, alleviating the memorization barrier at the data source, enables the conventional fine-tuning approach to yield a better code model. The results reveal a striking pattern: removing as little as the most memorized 10% of data yields a substantial improvement in Pass@1 on Eval-Human and Eval-Machine. Further, using fewer fine-tuning data points does not degrade generalization under Pass@k for large k. These suggest that memorized data act as a *memorization barrier*, capping the effectiveness of fine-tuning from the base model.

The memorization barrier reveals a new effect of memorization on the degraded generalization of LLM fine-tuning. This is different from prior studies, which have primarily examined its negative impact in the context of data contamination (Deng et al., 2024; Dong et al., 2024; Golchin and Surdeanu, 2025; Wang et al., 2025; Riddell et al., 2024), where test samples appear in the pretraining corpus, creating a false impression of "superior" generalization. To tackle this memorization barrier problem, dataset pruning based on memorization appears to be a straightforward way to improve code fine-tuning as shown in Fig. 4. However, this approach is *inefficient*: (1) it requires data-wise attribution to estimate memorization, which is difficult to scale since attribution itself is computationally intensive; and (2) determining the optimal pruning ratio for improved generalization is non-trivial, as exploring all possible choices requires multiple fine-tuning runs, making the process impractical. In the next section, we address the memorization barrier using the *information bottleneck* principle, which eliminates the need for additional data attribution and pruning.



(a) Eval-Human (b) Eval-Machine

Figure 4: Pass@k performance of the fine-tuned model on the OriGen dataset using DeepSeek-Coder-7b-Instruct-v1.5, evaluated by (a) *Eval-Human* and (b) *Eval-Machine* (Liu et al., 2023), with different pruning ratios of highly memorized fine-tuning data identified in Fig. 3. The x-axis shows the removal ratio of the most memorized examples (0% corresponds to full dataset), and the y-axis reports Pass@k for $k \in \{1, 5, 10\}$.

5 Information Bottleneck–Guided LLM Code Fine-tuning

IB regularization to overcome the memorization barrier. To address the memorization barrier identified in Sec. 4, we adopt the IB principle (Tishby et al., 2000; Lei et al., 2025; Liu et al., 2024b; Chen et al., 2024) as a remedy that requires no additional data attribution or pruning, enabling an automatic fine-tuning protocol without assuming prior knowledge of the barrier. Motivated by the analogy between the memorization barrier and the problem of *escaping bad local optima* (Ge et al., 2015; Criscitiello and Boumal, 2019) discussed in Sec. 4, we note that the latter can often be mitigated by *random re-initialization* in nonconvex optimization (Lee et al., 2016). Random initialization can be understood as restarting the optimization process, providing an equal opportunity to create a new optimization path and escape bad local optima.

Therefore, similar to random re-initialization, the rationale behind using IB is to transform differently memorized data into equally treated ones by compressing representations to discard spurious information while preserving task-relevant signals for prediction (Tishby et al., 2000). More concretely, the IB principle provides a framework for learning representations that are both predictive and compressed. Given input X, target Y, and representation Z, IB seeks Z that preserves task-relevant information about Y while discarding redundant

details from X. This is formulated as

$$\underset{p(z|x)}{\text{minimize}} \; I(X;Z) - \beta I(Z;Y), \tag{2}$$

where $p(z|x)$ gives the conditional distribution of the latent representation Z given the input X, and $I(\cdot;\cdot)$ denotes mutual information and $\beta > 0$ balances prediction against compression.

The compression term $I(X;Z)$ penalizes how much bottleneck representation Z memorizes input-specific details. In LLMs, direct computation is intractable, so we follow the variational IB framework (Alemi et al., 2017):

$$I(X;Z) \;\leq\; \mathbb{E}_{x\sim\mathcal{D}_{\text{code}}}\left[D_{\text{KL}}\big(q_{\boldsymbol{\phi}}(z|h_{\boldsymbol{\theta}}(x)) \,\|\, p(z)\big)\right], \tag{3}$$

where $\mathcal{D}_{\text{code}}$ denotes the dataset, D_{KL} is the Kullback–Leibler divergence, $p(z)$ is a simple prior (*e.g.*, $\mathcal{N}(0, I)$), $h_{\boldsymbol{\theta}}(x)$ denotes the hidden representation of input x extracted from a designated intermediate LLM layer (*e.g.*, layer 20), and $q_{\boldsymbol{\phi}}$ is the variational encoder (with the learnable parameters $\boldsymbol{\phi}$) that produces the representation Z given $h_{\boldsymbol{\theta}}(x)$. The design of $q_{\boldsymbol{\phi}}$ follows the standard variational IB setting (Alemi et al., 2017).

Minimizing this divergence discourages Z from encoding exact input patterns, thereby suppressing memorization of rare or spurious sequences. Hence, from (3), the compression loss is given by

$$\begin{aligned}&\ell_{\text{IB}}^{\text{compress}}(\boldsymbol{\theta},\boldsymbol{\phi})\\ =&\mathbb{E}_{x\sim\mathcal{D}_{\text{code}}}\left[D_{\text{KL}}\big(q_{\boldsymbol{\phi}}(z \mid h_{\boldsymbol{\theta}}(x)) \,\|\, p(z)\big)\right],\end{aligned} \tag{4}$$

In addition, the prediction term $I(Z;Y)$ in (2) prevents over-compression by ensuring that the bottleneck preserves information relevant for predicting Y. This term can be expressed by the log-likelihood of predicting Y from Z,

$$\ell_{\text{IB}}^{\text{predict}}(\boldsymbol{\theta},\boldsymbol{\phi}) = \mathbb{E}_{(x,y)\sim\mathcal{D}_{\text{code}}}\left[\log p_{\boldsymbol{\theta}}(y \mid z)\right], \tag{5}$$

where z is sampled from $q_{\boldsymbol{\phi}}(z|h_{\boldsymbol{\theta}}(x))$.

Integrating (4) with (5) based on (2), the proposed IB regularization loss is defined as

$$\ell_{\text{IB}}(\boldsymbol{\theta},\boldsymbol{\phi}) = \ell_{\text{IB}}^{\text{compress}}(\boldsymbol{\theta},\boldsymbol{\phi}) - \beta\ell_{\text{IB}}^{\text{predict}}(\boldsymbol{\theta},\boldsymbol{\phi}), \tag{6}$$

where β is an empirically chosen hyperparameter that balances the tradeoff between compression and prediction, consistent with the standard IB formulation. Furthermore, we integrate (6) with the standard fine-tuning objective in (1), yielding the **IB-regularized fine-tuning (IB-FT)**:

$$\underset{\boldsymbol{\theta},\boldsymbol{\phi}}{\text{minimize}} \;\; \ell_{\text{FT}}(\boldsymbol{\theta}) + \alpha\,\ell_{\text{IB}}(\boldsymbol{\theta},\boldsymbol{\phi}), \tag{IB-FT}$$

where ℓ_{FT} is the standard cross-entropy loss from (1), and $\alpha > 0$ controls the relative strength of the IB regularization.

Validating the effectiveness of IB through representation analysis. Recall that the motivation for using IB lies in its ability to compress representations while retaining predictive information, thereby reducing the influence of prediction-irrelevant (*i.e.*, spurious) features. This allows differently memorized data to be treated more uniformly during learning, mitigating the memorization barrier that arises from prior differences in memorization strength among code data.

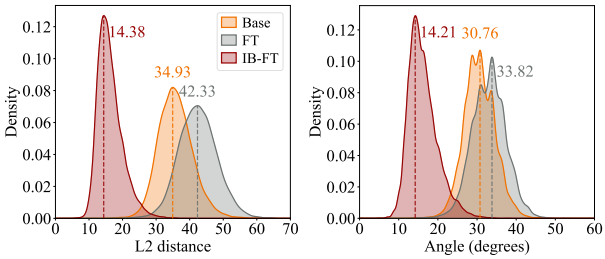


Figure 5: Representation distance and angle analysis with fine-tuning data on OriGen, using the base DeepSeek-Coder-7B-Instruct-v1.5 model and its fine-tuned variants (FT and IB-FT). Hidden representations $h_{\boldsymbol{\theta}}(x)$ are extracted from the 20th layer, and paired samples are drawn between the most-memorized and least-memorized groups (identified by `Min-20% Prob` scores in Fig. 3) to compute (a) their ℓ_2 distance distributions (left) and (b) angular disparity distributions (right). Dashed vertical lines mark distinct separation patterns under different fine-tuning regimes.

To validate the effectiveness of IB-FT, we analyze the representation geometry between memorized and non-memorized examples. Using `Min-K% Prob` scores, the dataset is partitioned into two subsets (most-memorized group and least-memorized group), and paired samples are drawn to measure ℓ_2 distances and angular disparities between their hidden representations at layer 20. As shown in **Fig. 5**, standard FT exaggerates the separation between the two groups relative to the base model, creating a barrier that forces the model to treat them in markedly different ways. In contrast, IB-FT substantially compresses this gap, reducing both distance and angular disparity by more than half. These results demonstrate that IB regularization yields more coherent, task-relevant representations across all data groups, thereby enabling fine-tuning to generalize more effectively.

6 Experiments

6.1 Experiment Setup

Datasets and models. Our experiments focus on two established datasets for LLM code fine-tuning:

Table 1: Performance for DeepSeek-Coder-7B-Instruct-v1.5 and CodeLlama-7B-Instruct fine-tuned on OriGen using IB-FT versus conventional fine-tuning (FT). Test-time accuracies are reported on the Eval-Human and Eval-Machine subsets using Pass@k ($k \in \{1, 5, 10\}$) and the stricter Pass@$10^{(m)}$ ($m \in \{2, 5, 10\}$), where Pass@$10^{(m)}$ counts a problem as solved only if at least m of the 10 generated samples pass the unit tests. Best results are shown in **bold**.

Methods	Eval-Human (%)						Eval-Machine (%)					
	Pass@k ↑			Pass@$10^{(m)}$ ↑			Pass@k ↑			Pass@$10^{(m)}$ ↑		
	$k=1$	$k=5$	$k=10$	$m=2$	$m=5$	$m=10$	$k=1$	$k=5$	$k=10$	$m=2$	$m=5$	$m=10$
DeepSeek-Coder-7B-Instruct-v1.5												
Base	34.62	42.09	44.23	42.31	36.54	24.36	56.47	64.76	67.13	63.64	58.74	43.66
+ FT	42.63	54.60	**57.77**	54.49	41.03	27.56	67.83	76.02	76.92	76.23	72.73	49.65
+ IB-FT	**51.99**	**54.86**	55.77	**54.49**	**51.92**	**48.07**	**75.67**	**78.51**	**79.02**	**78.32**	**77.62**	**69.23**
CodeLlama-7B-Instruct												
Base	17.56	27.02	29.49	26.28	17.95	4.19	44.76	53.79	55.24	53.85	47.55	27.27
+ FT	46.15	48.47	43.98	48.72	45.13	43.59	66.36	70.53	71.33	70.63	65.03	62.24
+ IB-FT	**49.68**	**52.82**	**46.81**	**51.28**	**50.64**	**45.51**	**70.07**	**72.94**	**73.47**	**72.73**	**70.63**	**64.33**

OriGen (Cui et al., 2024) for Verilog code generation, and **Evol-CodeAlpaca-V1** (Luo et al., 2024) for multi-language code generation. OriGen (Cui et al., 2024) is a Verilog instruction–response corpus with 222,075 examples, used to fine-tune LLMs for register-transfer-level (RTL) code generation. Evol-CodeAlpaca-V1 (Luo et al., 2024) is a multi-language instruction-to-code dataset of roughly 111,000 examples (covering Python, C++, Java, TypeScript, and Shell).

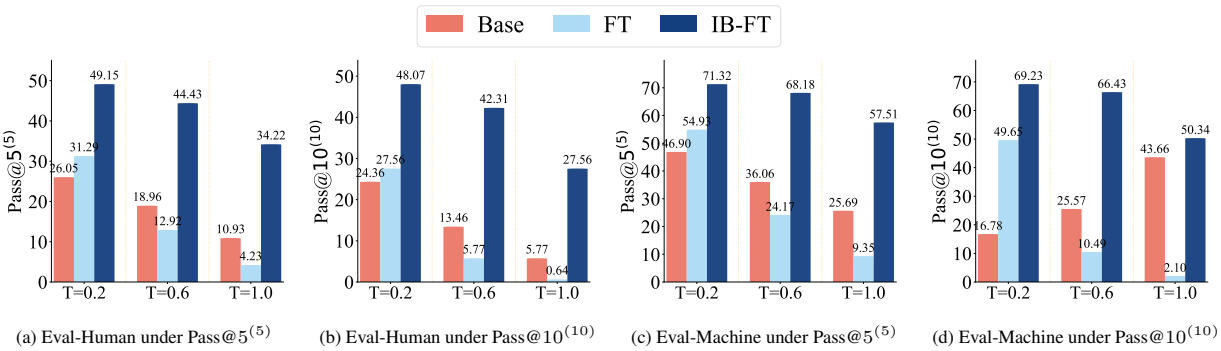
Following prior work (Cui et al., 2024; Luo et al., 2024), we adopt **DeepSeek-Coder-7B-Instruct-v1.5** (Guo et al., 2024) and **CodeLlama-7B-Instruct** (Roziere et al., 2023) as base models to validate the memorization barrier on the OriGen dataset. For Evol-CodeAlpaca-V1, we use **DeepSeek-Coder-7B-v1.5** (Guo et al., 2024) and **Llama-3-8B** (Grattafiori et al., 2024) as base models. And see **Appendix A.1** for more details about model fine-tuning parameters for FT and IB-FT.

Evaluation. For OriGen, we evaluate using the **VerilogEval** benchmark (Liu et al., 2023), which comprises two subsets: **Eval-Human** (human-authored problems, *e.g.*, HDLBits and manually transcribed tasks) and **Eval-Machine** (machine-generated prompts produced by LLMs). For Evol-CodeAlpaca-V1, we use **HumanEval** (Chen et al., 2021), a standard Python benchmark of 164 programming problems with natural-language descriptions, function signatures, and unit tests for automatic correctness evaluation. For evaluation metric, we introduce Pass@$k^{(m)}$, which requires at least m of k samples to succeed, offering a stricter and more reliable measure of generation consistency than standard Pass@k. More metric details shown in **Appendix A.2**.

6.2 Experiment Results

Performance overview of IB-regularized fine-tuning (IB-FT). In **Table 1** we present performance for LLMs (DeepSeek-Coder-7B-Instruct-v1.5 and CodeLlama-7B-Instruct) fine-tuned on OriGen using FT and our IB-FT, together with each model's pre-fine-tuning performance. Test-time accuracy is reported on the Eval-Human and Eval-Machine subsets using standard Pass@k ($k \in \{1, 5, 10\}$) and the stricter Pass@$10^{(m)}$ (fixed $k = 10$, $m \in \{2, 5, 10\}$).

As Table 1 shows, IB-FT consistently yields the top-rank performance: it outperforms both the base model and conventional FT across nearly all Pass@k settings, with especially large gains at $k = 1$. For example, on DeepSeek-Coder-7B-Instruct-v1.5 IB-FT achieves 51.99% (Eval-Human) and 75.67% (Eval-Machine) for Pass@1, versus 42.63% and 67.83% for FT. Increasing k (*e.g.*, to $k = 10$) narrows this gap, FT reaches 57.77% Pass@10 for DeepSeek-Coder-7B-Instruct-v1.5, which is better than IB-FT (55.77%). This suggests that some FT gains are realized only via sampling-based evaluation (Pass@k) rather than as reliable top-1 improvements, consistent with Fig. 2. Crucially, under the stricter Pass@$10^{(m)}$ criteria ($m > 1$), FT's scores fall off sharply, indicating that its successes are often isolated single samples rather than consistently reproducible generations. By contrast, IB-FT delivers more robust and reproducible improvements (notably at Pass@1 and



(a) Eval-Human under Pass@$5^{(5)}$ (b) Eval-Human under Pass@$10^{(10)}$ (c) Eval-Machine under Pass@$5^{(5)}$ (d) Eval-Machine under Pass@$10^{(10)}$

Figure 6: Performance of DeepSeek-Coder-7B-Instruct-v1.5 fine-tuned with IB-FT versus conventional FT and its base model (before fine-tuning) on OriGen across decoding temperatures $T \in \{0.2, 0.6, 1.0\}$. Results are reported on Eval-Human and Eval-Machine using Pass@$k^{(m)}$ with $m = k \in \{5, 10\}$. Other experimental setups follow Table 1.

Pass@$10^{(m)}$ for $m > 1$).

To stress, under the stricter Pass@$10^{(m)}$ metric, IB-FT substantially outperforms conventional FT. For the strictest setting ($m = 10$), which requires all 10 generations to pass, IB-FT improves over FT on DeepSeek-Coder-7B-Instruct-v1.5 by roughly 20% on both Eval-Human (*i.e.*, 48.07% by IB-FT vs. 27.56% by FT) and Eval-Machine (*i.e.*, 69.23% by IB-FT vs. 49.65% by FT). This shows that IB-FT yields stronger generalization and more stable multi-sample performance under demanding criteria, producing more reliable models in practice.

In addition, **Table A1** in **Appendix B.1** shows that IB-FT also consistently outperforms FT on the Evol-CodeAlpaca-V1 dataset. For example, on HumanEval, IB-FT improves Pass@1 from 56.70% to 61.60% for Llama-3-8B and from 51.28% to 54.57% for DeepSeek-Coder-7B-v1.5. Under the stricter Pass@$10^{(m)}$ setting, IB-FT further demonstrates stronger robustness, maintaining higher scores than FT across all m values. Also **Fig. A1** in **Appendix B.2** shows pruning memorized examples, though a natural remedy, is highly sensitive, whereas IB-FT is more robust.

Robustness of IB-FT under different temperature (T) settings. Next, we evaluate code generation after fine-tuning across decoding temperatures $T \in \{0.2, 0.6, 1.0\}$ (the default choice is $T = 0.2$). Varying T probes model robustness to sampling stochasticity and provides a more realistic assessment of fine-tuned performance and stability in practice. This protocol is motivated by the often-overlooked role of temperature in code model evaluation, since higher T yields more diverse generations and can materially affect Pass@k-style metrics (Wei et al., 2025; Cui et al., 2024).

Fig. 6 compares DeepSeek-Coder-7B-Instruct-v1.5 fine-tuned with IB-FT, the same model fine-tuned with conventional FT, and the pre-fine-tuned base model across decoding temperatures $T \in \{0.2, 0.6, 1.0\}$. Accuracy is measured by Pass@$k^{(m)}$ with $m = k \in \{5, 10\}$, *i.e.*, reflecting the all-generation success criterion if all k generated samples pass the unit tests. As we can see, IB-FT consistently sustains superior performance across all temperatures and benchmarks (Eval-Human and Eval-Machine), substantially outperforming both conventional FT and the base model even at high temperature. For example, on Eval-Human at $T = 1.0$, IB-FT attains 34.22% (Pass@$5^{(5)}$) versus 4.32% for FT; on Eval-Machine at $T = 1.0$, IB-FT reaches 50.34% while FT collapses to 2.10%. By contrast, FT shows poor robustness to temperature: it only exceeds the base model at the lowest temperature ($T = 0.2$) and suffers large drops as T increases. For instance, on Eval-Human at $T = 0.6$ FT falls to 12.92% (Pass@$5^{(5)}$) below the base model's 18.96%, and on Eval-Machine at $T = 1.0$ the base model retains 43.66% (Pass@$10^{(10)}$) while FT falls to 2.10%. These results indicate that IB-FT mitigates memorization-induced fragility and yields far more stable generalization under decoding stochasticity.

7 Conclusion

We identify a *memorization barrier* in LLM code fine-tuning: pretrained models often already memorize many downstream examples, which traps optimization and limits true adaptation and generalization. To address this, we introduce *IB-regularized fine-tuning* (IB-FT), which applies an information bottleneck (IB) penalty on hidden representations to suppress spurious memorized features while preserving task-relevant signals. Across OriGen and Evol-CodeAlpaca-V1, IB-FT yields consistently stronger and more stable gains, which position IB-FT as a practical, principled method for breaking memorization-driven limits in LLM adaptation.

8 Limitations

While our proposed IB-FT method effectively alleviates the memorization barrier in LLM code fine-tuning and surpasses state-of-the-art methods on two code generation benchmarks, some limitations remain. First, the introduced hyperparameters α (in (2)) and β (in IB-FT) are empirically selected. Their optimal values may depend on the underlying model architecture and dataset. Therefore, the sensitivity of hyperparameters in the proposed IB-FT should be further examined in future applications beyond those explored in this work. In addition, our experiments focus on code generation tasks and datasets. Although our proposed approach is theoretically general and could extend to other tasks, its performance beyond LLM code fine-tuning has not yet been verified.

References

Ali Al-Kaswan, Maliheh Izadi, and Arie Van Deursen. 2024. Traces of memorisation in large language models for code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–12.

Alexander A Alemi, Ian Fischer, Joshua V Dillon, and Kevin Murphy. 2017. Deep variational information bottleneck. In *International Conference on Learning Representations*.

Olav Beckmann, Alastair Houghton, Michael Mellor, and Paul HJ Kelly. 2004. Runtime code generation in c++ as a foundation for domain-specific optimisation. In *Domain-Specific Program Generation: International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003. Revised Papers*, pages 291–306. Springer.

Stella Biderman, Usvsn Prashanth, Lintang Sutawika, Hailey Schoelkopf, Quentin Anthony, Shivanshu Purohit, and Edward Raff. 2023. Emergent and predictable memorization in large language models. *Advances in Neural Information Processing Systems*, 36:28072–28090.

Nicholas Carlini, Daphne Ippolito, Matthew Jagielski, Katherine Lee, Florian Tramer, and Chiyuan Zhang. 2022. Quantifying memorization across neural language models. *arXiv preprint arXiv:2202.07646*.

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

Wentao Chen, Lizhe Zhang, Li Zhong, Letian Peng, Zilong Wang, and Jingbo Shang. 2025. Memorize or generalize? evaluating llm code generation with evolved questions. *arXiv preprint arXiv:2503.02296*.

Xin Chen, Hanxian Huang, Yanjun Gao, Yi Wang, Jishen Zhao, and Ke Ding. 2024. Learning to maximize mutual information for chain-of-thought distillation. In *ACL (Findings)*.

Christopher Criscitiello and Nicolas Boumal. 2019. Efficiently escaping saddle points on manifolds. *Advances in Neural Information Processing Systems*, 32.

Fan Cui, Chenyang Yin, Kexing Zhou, and 1 others. 2024. Origen: Enhancing rtl code generation with code-to-code augmentation and self-reflection. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, pages 1–9.

Simone Daniotti, Johannes Wachs, Xiangnan Feng, and Frank Neffke. 2025. Who is using ai to code? global diffusion and impact of generative ai. *arXiv preprint arXiv:2506.08945*.

Chunyuan Deng, Yilun Zhao, Xiangru Tang, Mark Gerstein, and Arman Cohan. 2024. Investigating data contamination in modern benchmarks for large language models. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 8698–8711.

Yihong Dong, Xue Jiang, Huanyu Liu, Zhi Jin, Bin Gu, Mengfei Yang, and Ge Li. 2024. Generalization or memorization: Data contamination and trustworthy evaluation for large language models. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 12039–12050.

Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, and Shuvendu K Lahiri. 2024. Llm-based test-driven interactive code generation: User study and empirical evaluation. *IEEE Transactions on Software Engineering*.

Rong Ge, Furong Huang, Chi Jin, and Yang Yuan. 2015. Escaping from saddle points—online stochastic gradient for tensor decomposition. In *Conference on learning theory*, pages 797–842. PMLR.

Shahriar Golchin and Mihai Surdeanu. 2025. Data contamination quiz: A tool to detect and estimate contamination in large language models. *Transactions of the Association for Computational Linguistics*, 13:809–830.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, and 1 others. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

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*.

Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33(8):1–79.

Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*.

Nam Huynh and Beiyu Lin. 2025. Large language models for code generation: A comprehensive survey of challenges, techniques, evaluation, and applications. *arXiv preprint arXiv:2503.01245*.

Paras Jain and 1 others. 2023. Swe-bench: Can language models resolve github issues? *NeurIPS*.

Hirokazu Kiyomaru, Issa Sugiura, Daisuke Kawahara, and Sadao Kurohashi. 2024. A comprehensive analysis of memorization in large language models. In *Proceedings of the 17th International Natural Language Generation Conference*, pages 584–596.

Hung Le and 1 others. 2022. Coderl: Mastering code generation through reinforcement learning with execution feedback. *NeurIPS*.

Jason D Lee, Max Simchowitz, Michael I Jordan, and Benjamin Recht. 2016. Gradient descent only converges to minimizers. In *Conference on learning theory*, pages 1246–1257. PMLR.

Shiye Lei, Zhihao Cheng, Kai Jia, and Dacheng Tao. 2025. Revisiting llm reasoning via information bottleneck. *arXiv preprint arXiv:2507.18391*.

Danny Leybzon and Corentin Kervadec. 2024. Learning, forgetting, remembering: Insights from tracking llm memorization during training. In *Proceedings of the 7th BlackboxNLP Workshop: Analyzing and Interpreting Neural Networks for NLP*, pages 43–57.

Kaixin Li, Qisheng Hu, Xu Zhao, Hui Chen, Yuxi Xie, Tiedong Liu, Qizhe Xie, and Junxian He. 2023. Instructcoder: Instruction tuning large language models for code editing. *arXiv preprint arXiv:2310.20329*.

Yawei Li, David Rügamer, Bernd Bischl, and Mina Rezaei. 2025. Calibrating LLMs with information-theoretic evidential deep learning. In *The Thirteenth International Conference on Learning Representations*.

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

Bingchang Liu, Chaoyu Chen, Zi Gong, Cong Liao, Huan Wang, Zhichao Lei, Ming Liang, Dajun Chen, Min Shen, Hailian Zhou, and 1 others. 2024a. Mft-coder: Boosting code llms with multitask fine-tuning. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5430–5441.

Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. 2023. Verilogeval: Evaluating large language models for verilog code generation. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 1–8. IEEE.

Zichuan Liu, Zefan Wang, Linjie Xu, Jinyu Wang, Lei Song, Tianchun Wang, Chunlin Chen, Wei Cheng, and Jiang Bian. 2024b. Protecting your llms with information bottleneck. *Advances in Neural Information Processing Systems*, 37:29723–29753.

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.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2024. Wizardcoder: Empowering code large language models with evol-instruct. In *ICLR*.

Yingwei Ma, Yue Liu, Yue Yu, Yuanliang Zhang, Yu Jiang, Changjian Wang, and Shanshan Li. 2023. At which training stage does code data help llms reasoning? *arXiv preprint arXiv:2309.16298*.

Pratyush Maini, Zhili Feng, Avi Schwarzschild, Zachary C. Lipton, and J. Zico Kolter. 2024. Tofu: A task of fictitious unlearning for llms. *Preprint*, arXiv:2401.06121.

Noble Saji Mathews and Meiyappan Nagappan. 2024. Test-driven development and llm-based code generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 1583–1594.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations*.

Martin Riddell, Ansong Ni, and Arman Cohan. 2024. Quantifying contamination in evaluating code generation capabilities of language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 14116–14137.

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

Andrew M Saxe, Yamini Bansal, Joel Dapello, Madhu Advani, Artemy Kolchinsky, Brendan D Tracey, and David D Cox. 2019. On the information bottleneck theory of deep learning. *Journal of Statistical Mechanics: Theory and Experiment*, 2019(12):124020.

Weijia Shi, Anirudh Ajith, Mengzhou Xia, Yangsibo Huang, Daogao Liu, Terra Blevins, Danqi Chen, and Luke Zettlemoyer. 2023. Detecting pretraining data from large language models. *arXiv preprint arXiv:2310.16789*.

Ravid Shwartz-Ziv and Naftali Tishby. 2017. Opening the black box of deep neural networks via information. *arXiv preprint arXiv:1703.00810*.

Shailja Thakur, Baleegh Ahmad, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Ramesh Karri, and Siddharth Garg. 2024. Verigen: A large language model for verilog code generation. *ACM Transactions on Design Automation of Electronic Systems*, 29(3):1–31.

Naftali Tishby, Fernando C Pereira, and William Bialek. 2000. The information bottleneck method. *arXiv preprint physics/0004057*.

Naftali Tishby and Noga Zaslavsky. 2015. Deep learning and the information bottleneck principle. In *2015 ieee information theory workshop (itw)*, pages 1–5. Ieee.

Chaozheng Wang, Yuanhang Yang, Cuiyun Gao, Yun Peng, Hongyu Zhang, and Michael R Lyu. 2022. No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence. In *Proceedings of the 30th ACM joint European software engineering conference and symposium on the foundations of software engineering*, pages 382–394.

Zeng Wang, Minghao Shao, Jitendra Bhandari, Likhitha Mankali, and 1 others. 2025. Vericontaminated: Assessing llm-driven verilog coding for data contamination. *arXiv preprint arXiv:2503.13572*.

Anjiang Wei, Huanmi Tan, Tarun Suresh, Daniel Mendoza, Thiago SFX Teixeira, Ke Wang, Caroline Trippel, and Alex Aiken. 2025. Vericoder: Enhancing LLM-Based RTL code generation through functional correctness validation. *arXiv preprint arXiv:2504.15659*.

Yuxiang Wei, Federico Cassano, Jiawei Liu, Yifeng Ding, Naman Jain, Zachary Mueller, Harm de Vries, Leandro Von Werra, Arjun Guha, and Lingming Zhang. 2024. Selfcodealign: Self-alignment for code generation. *Advances in Neural Information Processing Systems*, 37:62787–62874.

Frank F Xu, Bogdan Vasilescu, and Graham Neubig. 2022. In-ide code generation from natural language: Promise and challenges. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(2):1–47.

Zhou Yang, Zhengyu Qi, Zhaochun Ren, Zhikai Jia, Haizhou Sun, Xiaofei Zhu, and Xiangwen Liao. 2025. Exploring information processing in large language models: Insights from information bottleneck theory. *arXiv preprint arXiv:2501.00999*.

Zhou Yang, Zhipeng Zhao, Chenyu Wang, Jieke Shi, Dongsun Kim, Donggyun Han, and David Lo. 2024. Unveiling memorization in code models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13.

Shenglai Zeng, Yaxin Li, Jie Ren, Yiding Liu, Han Xu, Pengfei He, Yue Xing, Shuaiqiang Wang, Jiliang Tang, and Dawei Yin. 2023. Exploring memorization in fine-tuned language models. *arXiv preprint arXiv:2310.06714*.

Yujie Zhao, Hejia Zhang, Hanxian Huang, Zhongming Yu, and Jishen Zhao. 2024. Mage: A multi-agent engine for automated rtl code generation. *arXiv preprint arXiv:2412.07822*.

Terry Yue Zhuo, Armel Zebaze, Nitchakarn Suppattarachai, Leandro von Werra, Harm de Vries, Qian Liu, and Niklas Muennighoff. 2024. Astraios: Parameter-efficient instruction tuning code large language models. *arXiv preprint arXiv:2401.00788*.

Appendix

A Experiment Setup and Implementation Details

A.1 Fine-tuning Details

Base model. For reproducibility, we explicitly list the pretrained code foundation models used for fine-tuning on each dataset with link. On the OriGen dataset, we fine-tune DeepSeek-Coder-7B-Instruct-v1.5[2] and CodeLlama-7B-Instruct[3]. On the Evol-CodeAlpaca-V1 dataset, we fine-tune DeepSeek-Coder-7B-v1.5[4] and Meta-Llama-3-8B[5].

Fine-tuning setup for FT. For the baseline FT, we employ the LoRA (Low-Rank Adaptation) method (Hu et al., 2022). Across all datasets and models, we adopt the AdamW optimizer with $\beta_1 = 0.9$ and $\beta_2 = 0.999$, cosine learning rate decay, and a warm-up ratio of 0.03. Training is performed for 3 epochs with a batch size of 4, using mixed `float16` precision (while the base models were pretrained in `bfloat16`). The LoRA configuration is fixed to $r = 32$ and a dropout rate of 0.05, while α varies by dataset. On the OriGen dataset, training is conducted on two NVIDIA A100 GPUs with 80 GB memory, using an initial learning rate of 5×10^{-5} and LoRA $\alpha = 32$. On the Evol-CodeAlpaca-V1 dataset, training is conducted on four NVIDIA H100 GPUs, with an initial learning rate of 1×10^{-4} and LoRA $\alpha = 64$.

Fine-tuning setup for IB-FT. Our proposed IB-FT method is also built on the LoRA (Low-Rank Adaptation) framework. Across all datasets, we adopt the AdamW optimizer with $\beta_1 = 0.9$ and $\beta_2 = 0.999$, cosine learning rate decay, and a warm-up ratio of 0.03. Training is performed for 3 epochs with a batch size of 4, using mixed `float16` precision (while the base models were pretrained in `bfloat16`). For LoRA configuration, we set the rank $r = 32$ and the dropout rate to 0.05, while α varies depending on the dataset. On the OriGen dataset, IB-FT is trained on two NVIDIA A100 GPUs with 80 GB memory, using an initial learning rate of 5×10^{-5} and LoRA $\alpha = 32$. On the Evol-CodeAlpaca-V1 dataset, IB-FT is trained on NVIDIA H100 GPUs, with an initial learning rate of 1×10^{-4} and LoRA $\alpha = 64$.

In addition to the above configurations, IB-FT introduces two extra hyperparameters in Eq. (IB-FT): the IB regularization strengths α and β, which balance the trade-off between compression and prediction. We search over $\alpha \in (0.01, 1)$ and $\beta \in (0.001, 0.1)$. On the OriGen dataset, we set $\alpha = 0.1$ and $\beta = 0.02$ for both models, while on the Evol-CodeAlpaca-V1 dataset we use $\alpha = 0.1$ and $\beta = 0.01$.

A.2 Evaluation Metric

In addition to the standard $\text{Pass@}k$ metric, we adopt a stricter variant $\text{Pass@}k^{(m)}$ (as introduced in Fig. 2). Under $\text{Pass@}k^{(m)}$, a problem is counted as solved only if at least m out of the k generated samples pass the unit tests (so $\text{Pass@}k^{(1)}$ equals the conventional $\text{Pass@}k$). By requiring $m > 1$, $\text{Pass@}k^{(m)}$ penalizes lucky single-sample successes and provides a more reliable measure of a model's consistent generation quality and robustness. To evaluate this metric, we need to generate a pool of n candidate outputs for each problem (with $n \geq k$). Among these n outputs, the parameters k and m are then defined relative to this pool: k is the subset size under evaluation, and m is the minimum number of successful outputs required within that subset. In our experiments, we fix $n = 10$.

B Additional Experiment Results

B.1 Performance of Fine-tuning on Evol-CodeAlpaca-V1

We also evaluate Llama-3-8B and DeepSeek-Coder-7B-v1.5 on Evol-CodeAlpaca-V1, comparing conventional fine-tuning (FT), IB-FT, and the base model before fine-tuning. Evaluation is carried out on the HumanEval, using both the standard $\text{Pass@}k$ metric ($k \in 1, 5, 10$) and the more demanding $\text{Pass@}10^{(m)}$ criterion (with $k = 10$, $m \in 2, 5, 10$).

[2] https://huggingface.co/deepseek-ai/deepseek-coder-7b-instruct-v1.5
[3] https://huggingface.co/codellama/CodeLlama-7b-Instruct-hf
[4] https://huggingface.co/deepseek-ai/deepseek-coder-7b-base-v1.5
[5] https://huggingface.co/meta-llama/Meta-Llama-3-8B

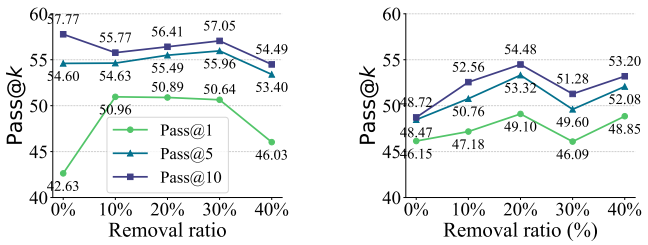
As **Table A1** shows, IB-FT consistently provides the strongest results across both models, with particularly notable improvements at $k = 1$. For instance, on Llama-3-8B, IB-FT achieves 61.60% for Pass@1 compared to 56.70% for FT, while on DeepSeek-Coder-7B-v1.5 the gains are 54.57% versus 51.28%. Moreover, under the strict $\text{Pass@10}^{(m)}$ setting, IB-FT demonstrates substantially higher robustness, reaching 37.20% at $m = 10$ on Llama-3-8B, which still provides a clear improvement over FT's 30.49%. These results highlight that IB-FT not only enhances top-1 accuracy but also yields more consistent and reproducible improvements under demanding multi-sample criteria, underscoring its advantage over conventional fine-tuning.

Table A1: Performance overview for Llama-3-8B and DeepSeek-Coder-7B-v1.5 fine-tuned on Evol-CodeAlpaca-V1 using IB-FT versus conventional fine-tuning (FT). Other metrics and table format follow those in Table 1.

Methods	HumanEval (%)					
	Pass@k ↑			$\text{Pass@10}^{(m)}$ ↑		
	$k=1$	$k=5$	$k=10$	$m=2$	$m=5$	$m=10$
Llama-3-8B						
Base	29.09	41.22	46.95	37.80	28.66	17.07
+ FT	56.70	74.16	79.27	73.17	62.80	30.49
+ IB-FT	**61.60**	**75.44**	**79.88**	**73.78**	**65.24**	**37.20**
DeepSeek-Coder-7B-v1.5						
Base	44.33	56.89	62.20	54.88	45.12	30.49
+ FT	51.28	64.10	64.10	63.46	50.00	32.05
+ IB-FT	**54.57**	**67.30**	**70.73**	**67.07**	**54.88**	**39.63**

B.2 Limitation of Pruning Memorized Examples, then Fine-tuning

Recall from Sec. 4 that removing highly-memorized examples (as characterized by `Min-K% Prob` scores) is a natural countermeasure to memorization. In **Fig. A1** we further examine this strategy on two base models (DeepSeek-Coder-7B-Instruct-v1.5 and CodeLlama-7B-Instruct) by removing 10%–40% of the most-memorized examples and reporting Pass@k for $k \in \{1, 5, 10\}$. The results reveal two sensitivity dimensions. First, the optimal removal ratio differs across models (*e.g.*, DeepSeek-Coder-7B-Instruct-v1.5 peaks at 10% for Pass@1 while CodeLlama-7B-Instruct peaks near 20%). Second, the optimal ratio varies with the metric even for the same model (*e.g.*, DeepSeek-Coder-7B-Instruct-v1.5's Pass@1 peaks at 10% whereas Pass@5 peaks at 30%). As shown, there is no universal pruning rate: removal is model- and metric-dependent. By contrast, our proposal avoids fragile data pruning and can be compatible with memorized examples during fine-tuning.



(a) DeepSeek-Coder-7B-Instruct-v1.5 (b) CodeLlama-7B-Instruct

Figure A1: Performance of fine-tuning with different data-removal ratios on OriGen, evaluated on the VerilogEval (Eval-Human) subset. Fine-tuning is applied to the base model (a) DeepSeek-Coder-7B-Instruct-v1.5 and (b) CodeLlama-7B-Instruct, and the performance is assessed using Pass@k ($k \in \{1, 5, 10\}$).