Inference-Time Techniques for Efficient Code Generation

Jacqueline He
jyyh@cs.washington.edu
University of Washington
Seattle, WA, USA

Melissa Mitchell
mcm08@cs.washington.edu
University of Washington
Seattle, WA, USA

Chung Yik Edward Yeung
chungy04@cs.washington.edu
University of Washington
Seattle, WA, USA

Shubham Tiwari
tshubham@cs.washington.edu
University of Washington
Seattle, WA, USA

Abstract

Large Language Models (LLMs) are frequently used by software developers for code generation, that is, the task of generating code snippets that adhere to specified natural language constraints. Accurate and reliable code generation is typically achieved by scaling the model size to billions of parameters, or training on trillions of tokens of code-specific data. Hence, modern state-of-the-art code LLMs tend to be proprietary (without open weights and only accessible via API, e.g., GitHub Copilot) or extremely large (i.e., around 100 billion parameters). Both the exorbitant cost and size of these models restrict developer accessibility and hinder the democratization of code LLM research; to this end, we propose to explore methods to improve code generation performance in small (around 1 billion parameters), open-weighted language models. In particular, we are interested in inference-time techniques, a broad class of *training-free* methods that boost model performance by spending extra compute during inference time.[1] We explore equipping small language models with two such techniques, context-aware decoding (CAD) and retrieval-augmented generation (RAG), and find that they lead to improved code generation abilities, thereby minimizing the performance gap against larger or proprietary models, while being cheaper and faster to run. We evaluate on MBPP and HumanEval, two popular, Python-based[2] code generation benchmarks. We show that CAD especially boosts performance, significantly increasing pass@10 scores (e.g., from 0.30 to 0.39 on HumanEval and from 0.40 to 0.50 on MBPP), while RAG provides complementary gains with minimal impact on throughput. Combined, these inference-time enhancements are observed to deliver up to 31% accuracy improvements over baseline methods with only a 13% reduction in throughput on certain comparisons, underscoring their potential to democratize access to high-quality code generation for resource-constrained developers.[3]

1 Introduction

Large language models (LLMs) show strong performance in domains such as code generation [7, 29]. However, many software developers and organizations cannot fully leverage large open models due to the steep infrastructure costs and operational challenges that scale linearly with their size [34, 4, 25]. This limitation hampers accessibility across a variety of domains: software teams with strict data governance requirements cannot rely on proprietary cloud-based solutions, while individual developers, small businesses, and academic researchers lack the computational resources to locally deploy a massive model. As a result, the potential of state-of-the-art LLMs remains unrealized for a significant user base.

Example use case. Consider a software engineer at a mid-sized healthcare startup who wants to integrate an LLM-driven code-generation feature for internal analytics scripts. Strict privacy regulations preclude sending patient data or generated code to external services. Although large open models (e.g., the largest LLaMA-3 model is 405B parameters) exist, the startup's on-premise GPU nodes may lack the necessary capacity to run the model efficiently[2]. Our project targets such scenarios by focusing on smaller, open-weight models and augmenting them with inference-time techniques to approximate or even rival the performance of larger LLMs on Python code generation tasks.

Closed-source LLMs such as GPT-4 and Codex [7] are state-of-the-art in code generation, often surpassing open-weight models [13]. But even open-weight models require significant resources to deploy (e.g., multiple high-end GPU nodes, higher VRAM) and incur significant operational overhead. For many organizations and individual developers, the scalability challenges—both in terms of compute cost and maintenance complexity—render these models impractical for day-to-day use, especially under stringent data governance [18] or budgetary constraints.

In contrast, smaller open LMs (3B parameters or less) take up 6GB of VRAM or less, and can be deployed on a single consumer GPU, leading to lower operational costs, simpler maintenance, and provide transparent model internals for security audits [34].[4] However, these models tend to underperform on popular code-generation benchmarks, producing outputs that contain malformed syntax or are more prone to logical errors than larger models. (While these benchmarks highlight the performance gap, the issue remains that such shortcomings directly affect the utility of smaller models in real-world coding tasks).

To this end, we hypothesize that inference-time techniques—training-free methods that allocate additional compute at test time—can bridge this performance gap.

In this work, we investigate two such techniques: context-aware decoding (CAD) and retrieval-augmented generation (RAG), which are commonly applied to knowledge-intensive tasks. Their utility

[1] Note that in this work, we use 'inference-time' and 'test-time' interchangeably.
[2] The scope of this project is restricted to Python-based code generation.
[3] Project code is released at: github.com/jacqueline-he/cse503.

[4] As a practical example, one can comfortably run a 3B model in half-precision using a standard 2023 MacBook Pro (M3 chip) with 8GB RAM.

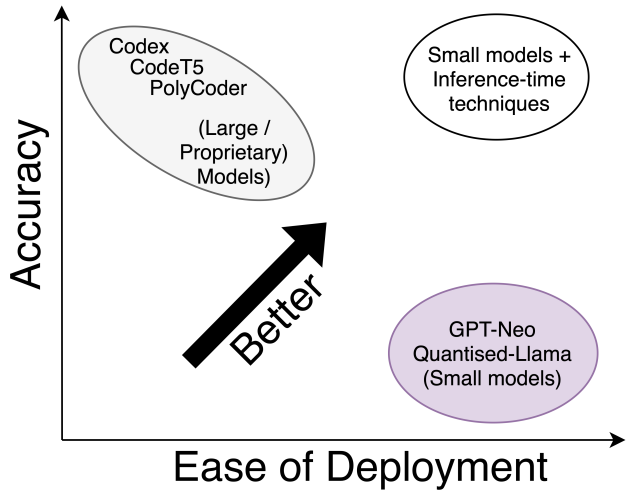

Figure 1: Trade-off between functional accuracy of code generation and ease of deployment. Large, proprietary models exhibit a high-accuracy in code-generation tasks. However they are not easy to train and deploy due to their size. Smaller models are easy to deploy but exhibit low accuracy in code-generation tasks. Proposed inference-time-toolkit (smaller models + inference-time techniques) would allow both ease of deployment and a significant improvement in accuracy.

to reasoning-heavy tasks (i.e., code generation) remains unexplored in the current literature.

Our experiments using a state-of-the-art 1B LM (Llama 3.2 1B) on two widely used Python-based evaluations, MBPP and HumanEval, yield several key insights:

- **CAD consistently boosts accuracy:** We identify CAD as a particularly robust inference-time technique that strictly improves code generation performance across the board. For instance, with Llama3.2 1B Instruct, CAD improves pass@10 on HumanEval from 0.30 to 0.39 and on MBPP from 0.40 to 0.50. CAD also strictly outperforms other inference-time baselines, namely chain-of-thought prompting (CoT) and self-consistency (SC). While combining CAD with RAG sometimes yields similar or slightly dampened improvements, CAD alone reliably enhances accuracy, although at the cost of roughly halving throughput.
- **RAG offers modest, efficient gains:** RAG preserves the vanilla model's throughput while offering minor accuracy improvements, albeit with sensitivity to retrieval quality. It performs similarly to Chain-of-Thought (CoT), marginally improving accuracy but also negligibly affecting throughput.
- **Inference-time techniques help improve the performance gap against larger models:** For example, while the vanilla Llama3.2 1B model achieves a pass@10 of 0.30 on HumanEval and 0.40 on MBPP, applying CAD boosts these numbers to 0.39 and 0.50, respectively. These results approach those of a much larger model—Gemma2 2B Instruct, which attains 0.41 on HumanEval and 0.50 on MBPP—demonstrating that inference-time methods can narrow the performance gap between smaller and larger models.
- **Inference-time techniques can surpass domain-specific training:** Notably, vanilla decoding with a code-specific model (StarCoder 1B) achieves a pass@10 of only 0.29 on HumanEval and 0.43 on MBPP. In contrast, inference-time techniques on top of a general-purpose, instruction-tuned model (Llama3.2 1B) consistently yield higher accuracy, with CAD reaching 0.39 on HumanEval and 0.50 on MBPP. This suggests that applying additional compute during inference can not only bridge but even surpass the performance achieved through domain-specific fine-tuning.

2 Related Work

2.1 LLMs for Code Generation

Modern LLMs demonstrate high accuracy across many automated code-related tasks, from code completion [19] to code repair [36] to code translation [9]. In particular, the task of *code generation*, which we define as the task of translating natural language into source code, has attracted significant interest beyond academic research, leading to the development of popular LLM-based coding assistants such as GitHub Copilot, CodeGeeX, and Amazon CodeWhisperer.

Enhancing an LLM's ability to generate syntactically correct and semantically meaningful code is primarily achieved in two ways: by pre-training a larger model on more general-purpose data, or by fine-tuning on specialized code datasets.

In the first way, [32] shows that general language modeling abilities scale smoothly (following an approximate power-law) with the number of model parameters and training data samples. Code generation is acquired *implicitly*, along with other general language abilities, during pre-training; large language models today encompass billions of parameters and are pre-trained on trillions of tokens in an unsupervised fashion, typically on web-scraped texts that include some code data. In a comprehensive study across multiple model families on the HumanEval benchmark, [37] shows that off-the-shelf code generation performance improves with increased model size. However, larger models require exponentially more floating point operations per second (FLOPS), memory, and parallel processing power during both training and inference. This means that usage of larger models is slower, more costly, and incurs a larger carbon footprint [10]. Another complication is that the largest LMs (i.e., > 100B parameters) tend to be closed-source to varying degrees. Some LLMs such as GPT-3 [6] or Codex [7] are only available as black-box APIs, and therefore cannot be trained.

A second way to improve code generation abilities without changing the model size is via a final stage of training on in-domain, high-quality code data [22]. For example, models such as CodeLlama [22], CodeT5 [29], PolyCoder [35], and Codex [7] are either trained from scratch on code-only data, or are further fine-tuned on code-only data. On general coding benchmarks, these code-specific models outperform their size-matched, domain-agnostic counterparts [37], underscoring the utility of domain-specific training. Yet one prominent trade-off to domain-specific training is the loss of generalization: the model may acquire more domain-specific

knowledge, at the cost of forgetting other general skills, a phenomenon known as catastrophic forgetting [16]. Furthermore, domain-specific fine-tuning requires careful curation of domain-specific data, which may rely on automated quality-filtering techniques that further require compute [1].

2.2 Inference-Time Techniques

Due to the compute and effort-intensive challenges that come with training language models, a line of research in *inference-time techniques* has recently become popular. Inference-time techniques do not require additional learning or weight updates, and operate only during the model's forward pass at test time to improve end task performance [8]. While inference time techniques boost end task performance in models of all size, smaller models generally experience larger *relative* gains. Therefore, one compelling aspect of spending more test-time computation is that it may allow a small model to match up to larger models; in a FLOPs-matched evaluation, [25] finds that careful allocation of inference-time compute can allow a small model to outperform a 14x larger model.

Chain-of-thought (CoT) prompting [14] and self-consistency (SC) prompting [28] are two simple inference-time techniques that have been shown to improve general *reasoning* performance, including code generation. In chain-of-thought prompting, the model is first asked to decompose a problem into a series of reasoning steps, which can later be verified for consistency. Self-consistency builds on this idea by generating multiple reasoning paths and selecting the most consistent output, thereby reducing variance and improving reliability. The idea behind self-consistency is that a single reasoning path may contain errors due to the LM's inherent variability; sampling multiple diverse reasoning paths and selecting the most consistent answer diminishes such a possibility.

As CoT and SC are widely used and known to boost code generation performance in language models, we consider them as comparison baselines.

Besides reasoning tasks, inference-time techniques have also been effective on knowledge-intensive tasks that require accurate factual recall. *Context-aware decoding* [24] mitigates unwanted factual hallucinations in generated output, by contrasting the probability distribution of candidate tokens when conditioned on a prompt witht context versus a version without context; this forces the model to adhere more closely to contextual knowledge. Additionally, Retrieval-Augmented Generation (RAG) [27] integrates external knowledge sources by retrieving relevant documents from a non-parametric datastore, thus enhancing the model's ability to ground itself on external text and generate more factual answers. Note that it is still an open question as to how these inference-time techniques would fare on reasoning-intensive tasks that do not require factual memorization. Therefore, we aim to explore their utility in code generation, a novel domain.

3 Proposed Approach

3.1 Language Models

We predominantly work with Llama-3.2-1B-Instruct [11], a state-of-the-art, small (i.e., 1.24 billion parameters), open-weighted, autoregressive language model that has undergone several phases of post-training to align with human behaviors [20]. Note that this model is not domain-specific, i.e., it has not undergone a training phase using exclusively code data, and that the training data composition is not disclosed. Given our time and compute constraints, we focus on models in the 1B size range.

3.2 CAD and RAG

Recent work has indicated that applying additional compute at inference time can benefit natural language tasks, resulting in performance gains *without* the need for additional training [25]. For example, prompting methods such as Chain-of-Thought [14] or Self-Consistency [18] have been found to enhance code generation performance. In this section, we explore the viability of two inference-time techniques, which have been originally proposed for knowledge-intensive tasks, on code generation.

3.2.1 Context-Aware Decoding (CAD). Context-aware decoding is a technique proposed by [24] that encourages greater prompt adherence, and has been shown to benefit context-reliant tasks, such as summarization or open-book question answering. The intuition behind CAD is that it increases context reliance by contrasting the probability distributions of output with and without context. Formally, assume a language model θ, an input instruction x, and some context c. The normal way to sample the output sequence y can be represented (at the t-th step) via

$$y_t \propto \exp \text{logit}_\theta(Y_t|c, x, y_{<t}). \quad (1)$$

CAD proposes to upweight reliance on the context c by factoring out prior knowledge from the model's context-less output distribution in a contrastive manner, like so:

$$y_t \propto \text{softmax}[(1+\alpha)\text{logit}_\theta(y_t|c, x, y_{<t}) - \alpha\text{logit}_\theta(y_t|x, y_{<t})], \quad (2)$$

wherein α is a controllable temperature term (and when set to 0, reduces to regular decoding). Note that CAD requires the LM to take two forward passes, so it is computationally twice as expensive as normal inference. We hypothesize that aside from natural language tasks, CAD may also benefit formal language tasks such as code; for example, on a coding completion task such as HumanEval, the LM ought to pay particular attention to the program prompt.

3.2.2 RAG. Retrieval-Augmented Generation (RAG) refers to a paradigm in which during inference time, input text is concatenated with semantically similar context retrieved from a static, external datastore, and passed to a language model (formally, the *reader*) [27].

The datastore is a non-parametric collection of document embeddings, and recent work has found that scaling the datastore can enable a smaller language model to surpass larger language models on a broad set of upstream and downstream knowledge-intensive tasks [23]. While RAG confers many benefits such as parameter efficiency, less hallucination, and the ability to provide attributions on knowledge-intensive tasks [27], it has not been widely applied to reasoning-intensive tasks such as code generation. A relatively new analysis on retrieval-augmented code generation [30], only tests LMs of a limited size range (i.e., $>$ 7B parameters) and datastores that are general-purpose or mixed-quality (i.e., scraped code from GitHub), but found only marginal improvements.

As our evaluation benchmarks are all Python-centric, we propose to experiment with RAG-assisted code generation by building and retrieving from a high-quality datastore of Python library documentation provided by `devdocs.io` following [38].[5] Intuitively, this is similar to how a human programmer might reference official code manuals or documentation when writing out a program. Following standard practice, we convert each document into document embeddings and build an index across all embeddings using the FAISS toolkit[6] and BGE, a state-of-the-art dense retriever.[7] Note that for our use case, it does not matter if the data used in the datastore overlaps with Llama 3.2 1B's training data. Even if the training data contains some of the documentation, retrieval ensures the model references the most relevant and correct details at generation time, thereby filling in gaps in the model's knowledge. We retrieve the top-k (where $k = 5$) documents for RAG.

3.2.3 Stacking inference-time approaches. Finally, we are interested in seeing if there is any additive performance benefit from merging our two proposed approaches (i.e., combining RAG with context-aware decoding); note that this is possible as these approaches operate at different stages of the LM generation pipeline.

Specifically, RAG augments the original input with additional context, while CAD directly modifies the models' decoding equation to force model reliance on context. Thus, we believe these two techniques would complement each other effectively (RAG + CAD).

4 Evaluation

In this section, we describe the evaluation datasets and metrics used, as well as baselines that we compare our proposed approaches against.

4.1 Datasets

We measure code generation performance using two popular code generation benchmarks, Mostly Basic Python Problems (MBPP) [3] and HumanEval [7]. Both MBPP and HumanEval encompass basic Python programming questions, and are typically evaluated by executing the generated code against a set of provided unit tests.

MBPP (test split) consists of 164 crowd-sourced basic Python programming problems, including topics such as fundamental programming and standard library functionality. Each problem is paired with three unit tests in the form of assert statements.

HumanEval (dev split) consists of 90 handwritten Python programming problems that test topics such as language comprehension, algorithms, and simple mathematics. Each problem includes a function signature, docstring, body, and unit tests to assess functional correctness.

4.2 Metrics

4.2.1 Functional correctness. For code generation, we adopt the pass@k metric [7] to measure the execution correctness of programs.

For the pass@k metric, k code samples are generated per problem. Each generation is executed as real code, and a problem is considered solved if any sample passes all the unit tests. The total fraction of problems solved is reported. Formally, with n as the total number of samples, and c as the number of correct samples,

$$\text{pass@k} := \mathbb{E}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right]. \tag{3}$$

Pass@k is a common evaluation metric for code generation. For one, [7] finds that pass@k is an unbiased estimator, which allows for fair comparison across different numbers of samples. Furthermore, match-based metrics such as BLEU and ROUGE, which were commonly used for code generation evaluation in the past, are insufficient at assessing functional correctness. Following standard practice, we report pass@k with k set to 1, 5, and 10.

4.2.2 Efficiency. Alongside functional correctness, we are also interested in evaluating the *efficiency* of model generations. Ideally, a small model that takes up additional inference-time compute should still generate more efficiently than a large model that uses vanilla generation.

We measure efficiency in terms of *throughput* (average number of tokens generated per second) and *memory overhead in MB* during token generation (the difference in VRAM before and after inference, using the PyTorch CUDA framework).

4.3 Baselines

First, we consider vanilla generation using Llama 3.2-1B-Instruct (i.e., without any inference-time modifications) as a baseline; this allows us to discern whether our proposed techniques show any improvement.

One question is how our proposed inference-time techniques–CAD and RAG–stack up against existing inference-time techniques that have been found to improve code generation. We compare both against chain-of-thought (CoT), a prompting technique that encourages models to explicitly think through each step.[8] Another prompting baseline is self-consistency (SC), in which instead of only generating once, the model generates multiple outputs to the same input (using a high temperature term, to encourage stochasticity). Typically, the most frequent answer is taken as the final answer (i.e., majority voting) if the output space is small (i.e., multiple choice options). However, in our case, we must choose between multiple long-form code generations, where exact matches are rare and frequency-based voting becomes unreliable. Therefore, we adapt SC by selecting the generation that is most consistent with the others, defined as the sequence with the highest average pairwise embedding similarity to the rest.

Another question is, how well can 1B models with our inference-time techniques compare against a model twice its size? This is especially interesting in the case of CAD, which requires two forward passes, and thus roughly doubles the inference cost per sample. Thus, an appropriate compute-matched comparison would be CAD using a 1B LM versus vanilla decoding using a 2B LM. Given that Llama 3 does not have a 2B model, we compare against Gemma 2

[5] Specifically, using this dataset of 34K samples: `huggingface.co/datasets/code-rag-bench/library-documentation`
[6] `https://github.com/facebookresearch/faiss`
[7] `https://huggingface.co/BAAI/bge-base-en`

[8] The simplest formulation of CoT is to append "`Let's think step by step`" to the prompt in a zero-shot manner (i.e., without any demonstrations), to encourage the model to first emit its reasoning process before arriving at the answer.

2B as a proxy [26].[9] Note that this does not constitute a perfectly fair comparison; however, Gemma and Llama are both architecturally similar (decoder-based Transformers) and likely pre-trained on similar data distributions (general Internet data).

A final question is, how well can 1B models with our inference-time techniques compare against size-matched models that have undergone domain-specific training? The fairest comparison would be vanilla generation with a CodeLlama 1B model (i.e., Llama- 3.2-1B-Instruct finetuned further on code data), which does not exist. As a proxy, we would compare against vanilla generation with StarCoder-1B [15] fine-tuned on code exercises,[10] a comparably-sized model that is instruction-tuned and has been trained on 80+ programming languages from permissively licensed code data.

5 Results

Table 1 and Table 2 show the full empirical results for HumanEval and MBPP, respectively, on functional accuracy (pass@k), throughput (average tokens generated per second), and memory overhead (the total change in VRAM). Our takeaways are as follows:

5.1 Empirical Takeaways

CAD consistently boosts accuracy: CAD achieves the best pass@k compared to the vanilla baseline, and the other inference-time baselines. On HumanEval, both CAD and CAD+RAG raise pass@10 from 0.30 (vanilla) to 0.39. On MBPP, CAD alone boosts pass@10 from 0.40 to 0.50, whereas CAD+RAG only reaches 0.48. This indicates that CAD reliably improves accuracy; however, adding the retrieval component (as in CAD+RAG) does not always yield additional benefits—in fact, on MBPP, it may slightly dampen the improvement observed with CAD alone. Notably, these accuracy gains come at a cost: approximately a 50% reduction in throughput.

CoT and RAG offer modest improvements with high efficiency: CoT yields moderate accuracy gains without affecting throughput, with more pronounced improvements observed at higher k-values (pass@5 and pass@10). Likewise, RAG maintains roughly the same throughput as the vanilla setting while delivering slight accuracy improvements in most cases. Its performance, however, is sensitive to the quality of retrieval; if the datastore fails to return relevant documents for a given query, irrelevant information may mislead the model. This problem is particularly salient if the datastore is small and retrieved context is *always* used, which is our case here. We hypothesize that with more sophisticated RAG techniques (i.e., selectively only using retrieved context if it is deemed relevant to the task) may lead to more discernable improvements.

SC delivers limited accuracy gains at a high computation cost: Despite some improvement in pass@k on HumanEval, SC does not largely improve over the vanilla baselines on MBPP (except on pass@5). However, SC dramatically reduces throughput by a factor of five (as we obtain 5 generations per sample).

Model size leads to markedly better accuracy, at the expense of speed and memory: Using a larger model achieves higher absolute accuracy, with the best pass@5 and pass@10 across the board for both MBPP and HumanEval. However, this comes at the expense of lower throughput (from 110 to 22 tokens / sec. for Humaneval, and 70 to 37 tokens / sec. for MBPP) and higher memory overhead (from 0.01 to 0.04 average MB for HumanEval, and 0.08 to 0.14 average MB for MBPP); the 2B LM achieves the worst throughput and memory overhead in all cases.

Some inference-time techniques, especially CAD, with a general-purpose model can generally surpass the performance of a code-specific model with vanilla decoding: Curiously, vanilla decoding with a code-specific LM (StarCoder 1B) does not lead to markedly higher pass@k compared to vanilla decoding with a size-matched general instruction-tuned LM (Llama 3.2 1B). We think this may be due to the nature of our prompts, which employ a lot of freeform natural language instructions (e.g., "Return only code for the completed function, with no additional explanation, print statements, or unit tests"), which Llama might be better-equipped to handle than StarCoder, which is primarily trained on structured code data. Compared to vanilla decoding with StarCoder 1B, CoT, CAD, and CAD+RAG on Llama 3.2 1B show strictly better improvement across both benchmarks.

5.2 Accuracy vs. Efficiency

Figure 2 and Figure 3 show visualizations of the accuracy vs. efficiency (throughput, memory overhead) trade-off for HumanEval and MBPP, respectively. Note that no setting unanimously dominates with the best accuracy, throughput, and memory.

Thus, the optimal balance between efficiency and pass rate is highly dependent on one's particular use case. For example, some techniques (e.g., CAD, or using a larger LM) may be preferable when accuracy is paramount, while others (e.g., RAG or CoT) might be preferred if maintaining throughput is critical.

6 Conclusion

We investigate the potential of inference-time techniques, to enhance code generation performance in small, 1B-scale language models, without the need for any training. Specifically, we study context-aware decoding (CAD) and retrieval-augmented generation (RAG) on two Python-based benchmarks, MBPP and HumanEval. Our work is the first to demonstrate that both CAD and RAG, which are commonly used on knowledge-intensive tasks, can also yield significant improvements in code generation (a new domain), with CAD showing particularly robust performance gains.

Our results suggest that inference-time techniques—especially CAD—can enable a general-purpose, instruction-tuned model (Llama 3.2 1B) to achieve competitive, and in some cases superior, performance relative to a size-matched model trained specifically on code. Moreover, these techniques can help narrow the performance gap between smaller and larger models, while offering advantages in terms of throughput and memory efficiency.

In light of these findings, we recommend that the community consider inference-time techniques as a viable alternative to extensive domain-specific fine-tuning or scaling up model sizes. Doing so could lead to more efficient and cost-effective deployments for code generation tasks, enabling improved performance without the need for larger models or additional training, and is a preliminary step towards democratizing LLMs.

[9] https://huggingface.co/google/gemma-2-2b-it
[10] `https://huggingface.co/jinaai/starcoder-1b-textbook`

Technique	pass@1 (↑)	pass@5 (↑)	pass@10 (↑)	Throughput (↑)	Memory Overhead (↓)	Base Model (and Size)
Vanilla	0.13	0.26	0.30	**110**	**0.01**	Llama 3.2 1B Instruct
CAD	0.17	0.31	0.39	58	**0.01**	Llama 3.2 1B Instruct
RAG	0.14	0.29	0.30	**110**	0.02	Llama 3.2 1B Instruct
CAD + RAG	0.17	0.32	0.39	56	0.02	Llama 3.2 1B Instruct
SC	0.12	0.28	0.34	22	**0.01**	Llama 3.2 1B Instruct
CoT	0.12	0.29	0.37	**110**	**0.01**	Llama 3.2 1B Instruct
Vanilla	**0.20**	**0.34**	**0.41**	22	0.04	Gemma 2 2B Instruct
Vanilla	0.16	0.26	0.29	100	**0.01**	StarCoder 1B

Table 1: Inference-time technique results on HumanEval. Best numbers for each metric are in bold.

Technique	pass@1 (↑)	pass@5 (↑)	pass@10 (↑)	Throughput (↑)	Memory Overhead (↓)	Base Model (and Size)
Vanilla	0.20	0.35	0.40	70	**0.08**	Llama 3.2 1B Instruct
CAD	0.23	0.46	**0.50**	39	**0.08**	Llama 3.2 1B Instruct
RAG	0.17	0.38	0.42	**71**	0.09	Llama 3.2 1B Instruct
CAD + RAG	**0.24**	0.47	0.48	38	0.09	Llama 3.2 1B Instruct
SC	0.19	0.38	0.40	14	**0.08**	Llama 3.2 1B Instruct
CoT	0.21	0.38	0.41	72	**0.08**	Llama 3.2 1B Instruct
Vanilla	0.19	**0.48**	**0.50**	37	0.14	Gemma 2 2B Instruct
Vanilla	0.19	0.36	0.43	69	**0.08**	StarCoder 1B

Table 2: Comparison of inference-time techniques on MBPP. Best numbers for each metric are in bold.

7 Limitations

There are several limitations to our work. To begin, our evaluation is restricted to two Python-based benchmarks (MBPP and HumanEval). Python is a popular, high-resourced programming language with a readable and intuitive syntax that is similar to natural language. Python is also the dominant language of many pre-training code datasets, such as The Stack [17]. Therefore, it is unknown whether our findings can generalize to lower-resourced languages, or ones with more complex syntax. Furthermore, both benchmarks would not have as much real-world applicability in more sophisticated instances that involve larger codebase reasoning, long-horizon debugging, or other practical considerations such as efficiency, security, readability, or maintainability.

Recent studies [5, 21] have surfaced the possibility of *benchmark contamination*, in which potential data leakage from common evaluation sets into the training data of the language model may artificially boost its code generation performance. Furthermore, the models we use are all open-weight, but not entirely open-sourced (i.e., the training pipeline and datasets are not publicly released), meaning that there is a non-negligible risk of evaluation contamination, and that model performance could potentially be partially attributed to the presence of evaluation data in training data. This concern is especially salient as MBPP and HumanEval are popular evaluation datasets.

However, a recent systematic study [21] finds that Llama 3 (the model family we mainly use), which is trained on larger and better decontaminated data, shows limited signs of test set leakage across multiple coding evaluations. Furthermore, contamination is also less of a problem due to the small models we use. Memorization is an ability that scales with model capacity, and so larger models are better at storing and recalling examples verbatim from training data. Therefore, while benchmark contamination is a general concern, it is less significant in our case due to our choice in model family and model size.

Finally, while our findings are promising, our experiments are non-exhaustive, primarily due to compute budget reasons. We leave a more comprehensive and varied baseline evaluation to future work. For example, we use Llama 3.2 1B Instruct as our base model, as it is small enough to run on a single 40GB GPU. While it is the most fair and meaningful to compare inference-time techniques against the vanilla baseline for the **same base model**, we had to use models from different families (i.e., Gemma 2B and StarCoder 1B) as the Llama-specific versions (a Llama 2B model or a CodeLlama 1B model) do not exist.

In order for our findings to be truly generalizable, we would have to additionally evaluate on models from other model families and across different parameter scales (i.e., 8B, 13B, 70B). For our findings to be robust and statistically significant, we would need to do multiple runs across different random seeds for each setting. We also think our study could benefit from a more exhaustive ablation across various design choices, for example, different decoding temperatures, different values of α for CAD, different choices in datastore composition for RAG (such as actual code instead of official Python documentation), or different prompting formats.

Finally, our inference-time baselines (CoT and SC) are simple and employ very straightforward techniques (prompting and sampling). Future work can compare CAD and RAG across current state-of-the-art inference-time techniques that employ backtracking, domain knowledge, or refinement algorithms [33].

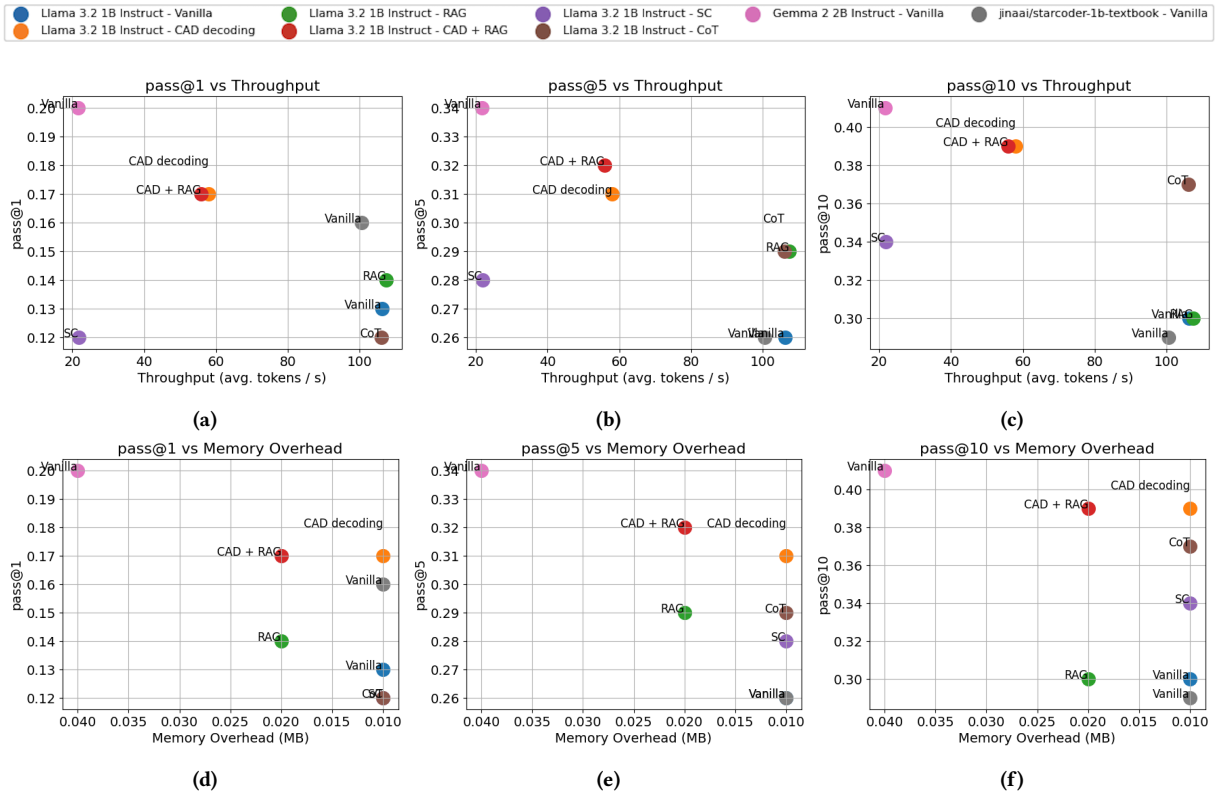

Figure 2: Accuracy vs. Throughput (first row) and Accuracy vs. Memory Overhead (second row) trade-offs for inference-time techniques on HumanEval. Note that the x-axes are inverted for memory overhead. In all figures, positive direction of the line y=x is direction (high accuracy, high efficiency) with the best trade-off.

References

[1] Amro Abbas, Kushal Tirumala, Dániel Simig, Surya Ganguli, and Ari S. Morcos. 2023. Semdedup: data-efficient learning at web-scale through semantic deduplication. (2023). https://arxiv.org/abs/2303.09540 arXiv: 2303.09540 [cs.LG].

[2] Reza Yazdani Aminabadi et al. 2022. Deepspeed inference: enabling efficient inference of transformer models at unprecedented scale. (2022). https://arxiv.org/abs/2207.00032 arXiv: 2207.00032 [cs.LG].

[3] Jacob Austin et al. 2021. Program synthesis with large language models. (2021). https://api.semanticscholar.org/CorpusID:237142385.

[4] Sid Black, Leo Gao, Phil Wang, Connor Leahy, and Stella Biderman. 2021. Gpt-neo: large scale autoregressive language modeling with mesh-tensorflow. In https://api.semanticscholar.org/CorpusID:245758737.

[5] Jeremy S. Bradbury and Riddhi More. 2024. Addressing data leakage in humaneval using combinatorial test design. *ArXiv*, abs/2412.01526. https://api.semanticscholar.org/CorpusID:274436590.

[6] Tom B. Brown et al. 2020. Language models are few-shot learners. (2020). https://arxiv.org/abs/2005.14165 arXiv: 2005.14165 [cs.CL].

[7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, et al. 2021. Evaluating large language models trained on code. (2021). https://arxiv.org/abs/2107.03374 arXiv: 2107.03374 [cs.LG].

[8] Xiangjue Dong, Maria Teleki, and James Caverlee. 2024. A survey on llm inference-time self-improvement. (2024). https://arxiv.org/abs/2412.14352 arXiv: 2412.14352 [cs.CL].

[9] Hasan Ferit Eniser, Hanliang Zhang, Cristina David, Meng Wang, Maria Christakis, Brandon Paulsen, Joey Dodds, and Daniel Kroening. 2024. Towards translating real-world code with llms: a study of translating to rust. (2024). https://arxiv.org/abs/2405.11514 arXiv: 2405.11514 [cs.SE].

[10] Ahmad Faiz, Sotaro Kaneda, Ruhan Wang, Rita Osi, Prateek Sharma, Fan Chen, and Lei Jiang. 2024. Llmcarbon: modeling the end-to-end carbon footprint of large language models. (2024). https://arxiv.org/abs/2309.14393 arXiv: 2309.14393 [cs.CL].

[11] Aaron Grattafiori et al. 2024. The llama 3 herd of models. (2024). https://arxiv.org/abs/2407.21783 arXiv: 2407.21783 [cs.AI].

[12] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. The curious case of neural text degeneration. In *International Conference on Learning Representations*. https://openreview.net/forum?id=rygGQyrFvH.

[13] Wenpin Hou and Zhicheng Ji. 2024. Comparing large language models and human programmers for generating programming code. *Advanced Science*, (Dec. 2024). doi:10.1002/advs.202412279.

[14] Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2023. Structured chain-of-thought prompting for code generation. (2023). https://arxiv.org/abs/2305.06599 arXiv: 2305.06599 [cs.SE].

[15] Raymond Li et al. 2023. Starcoder: may the source be with you! arXiv: 2305.06161 [cs.CL].

[16] Chengyuan Liu, Yangyang Kang, Shihang Wang, Lizhi Qing, Fubang Zhao, Changlong Sun, Kun Kuang, and Fei Wu. 2024. More than catastrophic forgetting: integrating general capabilities for domain-specific llms. (2024). https://arxiv.org/abs/2405.17830 arXiv: 2405.17830 [cs.CL].

[17] Anton Lozhkov et al. 2024. Starcoder 2 and the stack v2: the next generation. (2024). https://arxiv.org/abs/2402.19173 arXiv: 2402.19173 [cs.SE].

[18] Marcus J. Min, Yangruibo Ding, Luca Buratti, Saurabh Pujar, Gail Kaiser, Suman Jana, and Baishakhi Ray. 2024. Beyond accuracy: evaluating self-consistency of code large language models with identitychain. (2024). https://arxiv.org/abs/2310.14053 arXiv: 2310.14053 [cs.LG].

[19] Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2024. Is self-repair a silver bullet for code generation? In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=y0GJXRungR.

[20] Long Ouyang et al. 2022. Training language models to follow instructions with human feedback. (2022). https://arxiv.org/abs/2203.02155 arXiv: 2203.02155 [cs.CL].

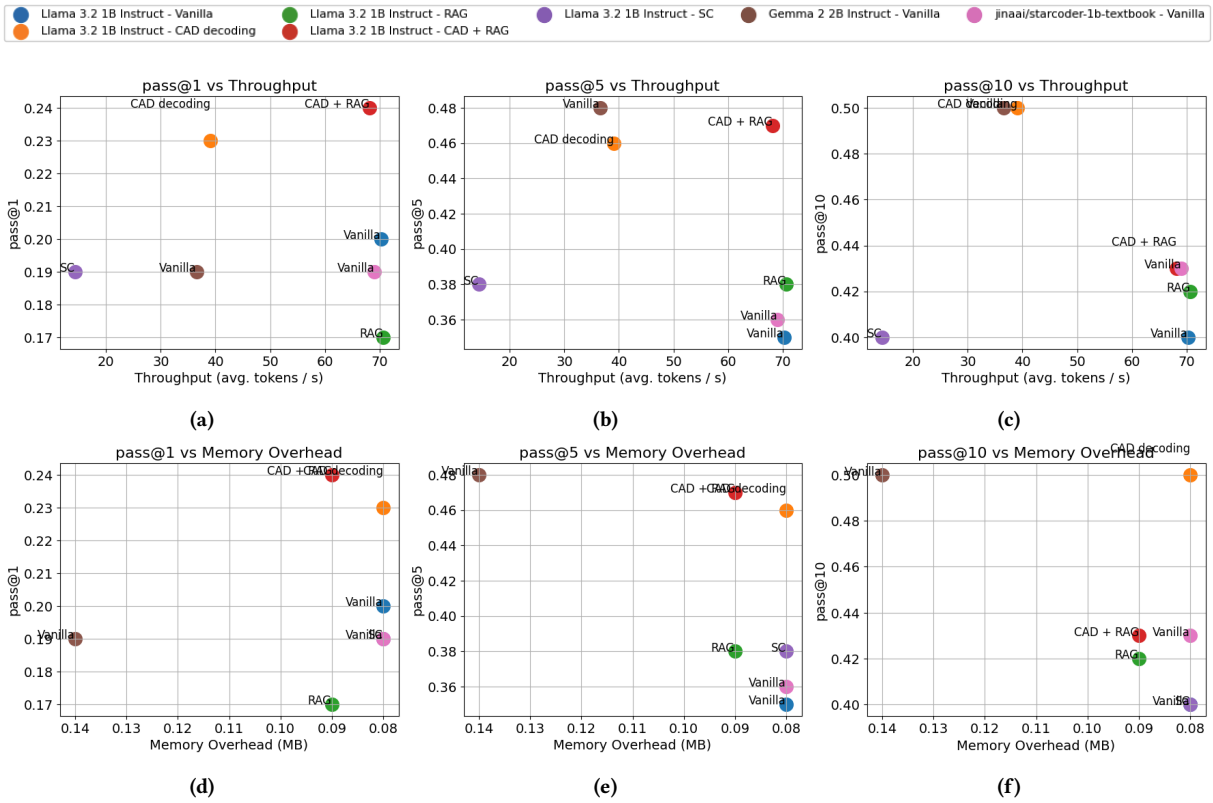

Figure 3: Accuracy vs. Throughput (first row) and Accuracy vs. Memory Overhead (second row) trade-offs for inference-time techniques on MBPP. Note that the x-axes are inverted for memory overhead. In all figures, positive direction of the line y=x is direction (high accuracy, high efficiency) with the best trade-off.

[21] Daniel Ramos, Claudia Mamede, Kush Jain, Paulo Canelas, Catarina Gamboa, and Claire Le Goues. 2024. Are large language models memorizing bug benchmarks? *ArXiv*, abs/2411.13323. https://api.semanticscholar.org/CorpusID:274150550.

[22] Baptiste Rozière et al. 2024. Code llama: open foundation models for code. (2024). https://arxiv.org/abs/2308.12950 arXiv: 2308.12950 [cs.CL].

[23] Rulin Shao, Jacqueline He, Akari Asai, Weijia Shi, Tim Dettmers, Sewon Min, Luke Zettlemoyer, and Pang Wei Koh. 2024. Scaling retrieval-based language models with a trillion-token datastore. (2024). https://arxiv.org/abs/2407.12854 arXiv: 2407.12854 [cs.CL].

[24] Weijia Shi, Xiaochuang Han, Mike Lewis, Yulia Tsvetkov, Luke Zettlemoyer, and Scott Wen-tau Yih. 2023. Trusting your evidence: hallucinate less with context-aware decoding. (2023). https://arxiv.org/abs/2305.14739 arXiv: 2305.14739 [cs.CL].

[25] Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. 2024. Scaling llm test-time compute optimally can be more effective than scaling model parameters. (2024). https://arxiv.org/abs/2408.03314 arXiv: 2408.03314 [cs.LG].

[26] Gemma Team et al. 2024. Gemma 2: improving open language models at a practical size. (2024). https://arxiv.org/abs/2408.00118 arXiv: 2408.00118 [cs.CL].

[27] Gemma Team et al. 2024. Gemma: open models based on gemini research and technology. (2024). https://arxiv.org/abs/2403.08295 arXiv: 2403.08295 [cs.CL].

[28] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-consistency improves chain of thought reasoning in language models. (2023). https://arxiv.org/abs/2203.11171 arXiv: 2203.11171 [cs.CL].

[29] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. 2021. Codet5: identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. (2021). https://arxiv.org/abs/2109.00859 arXiv: 2109.00859 [cs.SE].

[30] Zora Z. Wang, Akari Asai, Xinyan V. Yu, Frank F. Xu, Yiqing Xie, Graham Neubig, and Daniel Fried. 2024. Coderag-bench: can retrieval augment code generation?

[31] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-thought prompting elicits reasoning in large language models. (2023). https://arxiv.org/abs/2201.11903 arXiv: 2201.11903 [cs.CL].

[32] Jason Wei et al. 2022. Emergent abilities of large language models. (2022). https://arxiv.org/abs/2206.07682 arXiv: 2206.07682 [cs.CL].

[33] Sean Welleck, Amanda Bertsch, Matthew Finlayson, Hailey Schoelkopf, Alex Xie, Graham Neubig, Ilia Kulikov, and Zaid Harchaoui. 2024. From decoding to meta-generation: inference-time algorithms for large language models. (2024). https://arxiv.org/abs/2406.16838 arXiv: 2406.16838 [cs.CL].

[34] Frank F. Xu, Uri Alon, Safwan Hossain, Graham Neubig, and Vincent J. Hellendoorn. 2022. A systematic evaluation of large language models of code. (2022). https://arxiv.org/abs/2202.13169 arXiv: 2202.13169 [cs.CL].

[35] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent J. Hellendoorn. 2022. A systematic evaluation of large language models of code. (2022). https://arxiv.org/abs/2202.13169 arXiv: 2202.13169 [cs.PL].

[36] Junjielong Xu, Ying Fu, Shin Hwei Tan, and Pinjia He. 2025. Aligning the objective of llm-based program repair. (2025). https://arxiv.org/abs/2404.08877 arXiv: 2404.08877 [cs.SE].

[37] Daoguang Zan, B. Chen, Fengji Zhang, Di Lu, Bingchao Wu, Bei Guan, Yongji Wang, and Jian-Guang Lou. 2022. Large language models meet nl2code: a survey. In *Annual Meeting of the Association for Computational Linguistics*. https://api.semanticscholar.org/CorpusID:258557362.

[38] Shuyan Zhou, Uri Alon, Frank F. Xu, Zhiruo Wang, Zhengbao Jiang, and Graham Neubig. 2023. Docprompting: generating code by retrieving the docs. (2023). https://arxiv.org/abs/2207.05987 arXiv: 2207.05987 [cs.CL].

A Appendix

A.1 Decoding details

Across all decoding settings, we employ nucleus sampling [12] (a probabilistic decoding strategy that balances fluency and diversity) with a top-p of 0.9, a temperature of 0.8, and a maximum sequence length of 128.

Generation was quite slow as we used HuggingFace instead of speed-optimized inference frameworks such as `vllm`; we could not use the latter as modifying it to support CAD would take substantial work. Furthermore, we could only support single-GPU runs due to compute constraints (some of us only had access to Google Colab). Adapting to a distributed inference setting in future work could definitely speed up the iteration of our experiments.

We run evaluation for both benchmarks in a multiprocessed fashion using the pass@k code provided by OpenAI's HumanEvaluation package.[11]

A.2 Inference-time technique implementation details

A comparison of the four inference-time techniques used in this paper (CAD, RAG, CoT, SC) are in Table 3. Prompts for each setting are in Table 4.

Further details for each inference-time method are provided below.

CAD. We implemented Context-Aware Decoding following code from [24], setting $\alpha = 0.5$. CAD requires a forward passes of the model across two input sequences, the *context* input (which is the entire prompt, including instruction, function docstring, and function signature) and the *question* input (which is just the function signature). Note that CAD is the only technique that changes the generation equation; all other techniques use standard decoding.

RAG. We constructed a datastore using Python library documentation. Following standard practice [23], we split documentation into 500-word (as determined by whitespace) and append the documentation title (e.g., `cmath.isclose`) to the beginning of each chunk. We use the state-of-the-art BGE retriever released by BAAI. In total, our datastore consists of 44007 chunks.

Furthermore, we performed offline retrieval (with 5 docs per sample) on the two evaluation sets, MBPP and HumanEval, using the context for each evaluation's test set as the retrieval query. Due to time and compute constraints, we only evaluate with the top-1 document for the RAG setting.

CoT. Following [31], we instruct the LM to first think step-by-step about the solution before generating the rest of the function.

SC. We implement self-consistency by generating 5 sequences per evaluation sampling, using the same prompt as vanilla decoding, and a relatively high temperature (0.8) for sampling diversity. One problem in our case is that generations are long-form, so it is difficult to find the most consistent generation out of five with a simple string equality check (unlike, say, a particular task in which the possible set of outputs is very small). Therefore, we identify the most consistent generation by computing pairwise embedding similarities and using the output with the highest total similarity. We encode generations with a small sentence embedding Transformer model, `"all-mpnet-base-v2"`.[12]

[11] `https://github.com/openai/human-eval`

[12] `https://huggingface.co/sentence-transformers/all-mpnet-base-v2`

Technique	Multiple Forward Passes?	External Knowledge?	Vanilla Decoding?	Modification
CAD	✓(2 passes)	✗	✗	Modifies decoding algorithm
RAG	✗	✓	✓	Augments input prompt
SC	✓(5 passes)	✗	✓	Samples multiple outputs
CoT	✗	✗	✓	Augments input prompt

Table 3: Comparison of inference-time techniques for improving code generation.

Setting	Input Prompt
Vanilla, SC	`"Your task is to complete a Python programming problem, given its function signature and docstring. Return only code for the completed function, with no additional explanation, print statements, or unit tests. \n\n {prompt}"`
CAD	`"Your task is to complete a Python programming problem, given its function signature and docstring. Return only code for the completed function, with no additional explanation, print statements, or unit tests. \n\n Context: {function_docstring} \n {function_signature}"`
RAG	`"Your task is to complete a Python programming problem, given its function signature and docstring. Return only code for the completed function, with no additional explanation, print statements, or unit tests. Here is some context that may be helpful: {rag_doc} \n\n {prompt}"`
CoT	`Your task is to accurately complete a Python programming problem, given a function description. Before completing the function, think step by step about the solution. First, outline the approach based on the function signature and docstring. Break down the problem, identify edge cases, and describe the logic needed. Then, implement the function accordingly. Return only code for the completed function, with no additional explanation, print statements, or unit tests.\n{prompt}\n\n"`

Table 4: Prompt formats for different settings. The `{prompt}` is provided by the evaluation set. Note that for CAD, we decomposed `{prompt}` into `{function_docstring}` and `{function_signature}`. For RAG, we feed in the top-1 retrieved text as `{rag_doc}`.