Program Selection Using Code Execution Agents

Raunak Sood, Grant Ovsepyan, Chyi-Jiunn Lin
Advanced NLP
Carnegie Mellon University
{rrsood, govsepya, chyijiul}@andrew.cmu.edu

Abstract

Large Language Models (LLMs) have shown remarkable results on code generation tasks such as natural language to program synthesis, code infilling, and test-case synthesis. Despite this recent success, selecting a single correct program from multiple candidates generated by an LLM remains a hard problem. Many state-of-the-art models perform significantly worse when they are given a single opportunity to produce a program (pass@1) compared to when they are given multiple chances (pass@k) (Li et al., 2022). In this paper, we survey the current state-of-the-art methods aimed at resolving this issue, re-implement the most popular of them (CodeT) (Chen et al., 2022), and report the baselines we achieve. We also point out key flaws in CodeT and it's reliance on randomly generated test cases, which can lead to suboptimal program selection,. This is why we propose a new method for more accurate program selection, which we plan on implementing for the final project.

1 Introduction

In natural language (NL) to program synthesis, a user describes a target program via an NL description of the task, and the synthesizer finds a program that is consistent with it. Here, consistency means that the program does the task corresponding to the description. This has many applications in the software industry as developers can quickly generate code based on their desired specifications without much effort.

However, choosing a single correct program from a generated set for each problem is challenging. One of significant challenges in accurately selecting the correct program from multiple candidates generated by LLMs is in the cases when models assign high probabilities to syntactically plausible yet functionally incorrect code. For instance, suppose we provide an LLM with a description of the function that returns the square of an input number. In generation, the model might assign high probability to the correct function, namely $f(x) = x^2$. However, it might also assign a high probability to incorrect solutions, say $f(x) = 2x$, since LLM decoding is based on the log-likelihood of the generated tokens, instead of information regarding functional correctness. Thus, when we sample from the model, we want a method to avoid selecting these types of incorrect solutions.

Our work focuses on addressing this challenge by exploring methods to improve the selection of correct programs, aiming to enhance the efficiency and reliability of LLMs in code generation tasks. One relaxation of this problem is allowing model's to generate multiple programs and submitting all of them as the solution to the problem. Then, if any of these solutions are valid, we mark the generations as valid. This is known as the pass@k score of a model where k is the number of attempts the model is allowed. In many cases, we observe a log-linear relationship between k and the pass@k score (Li et al., 2022). This implies that as the number of programming solutions a model generates increases, the likelihood that one of these programs is correct also increases up to a certain threshold. The goal in program synthesis is thus to increase the slope of this curve, which would indicate that we need fewer submissions to generate the correct program.

2 Background

Current methodologies exhibit notable limitations in their approaches. While MBR-Exec considers test input execution, it underutilizes output validation in its assessment criteria. Reflexion introduces linguistic feedback mechanisms but falls short in prioritizing challenging test cases. CodeT attempts to address these limitations through dual-execution agreement, however as we noted, our analysis reveals fundamental constraints in its test case selection strategy, which our proposed methodology seeks to resolve.

Re-ranking model-generated programs is the most common way of improving pass@k score. In this set up, we prompt the model to generate k programs ranked based on log-likelihood and use a re-ranking algorithm to place better programs first. In this section, we describe the current state-of-the-art algorithms for re-ranking programs based on test-execution agreement. Out of these algorithms, we re-implement CodeT (Chen et al., 2022) in the rest of the report and plan on using this algorithm for our final project.

2.1 MBR-Exec

Execution-based minimum Bayes risk decoding (MBR-Exec) uses a similar strategy as CodeT; however, it relies on only test-case inputs as opposed to both inputs and outputs (Shi et al., 2022). The algorithm first samples a set of programs $\mathcal{P}$ and test cases $\mathcal{T}$ from an LLM based on a natural language description. Then, we execute each program on each test case and select the program whose execution results differ from the least number of other sampled programs. In other words, we select a program

$$\hat{p} = \text{argmin}_{p \in \mathcal{P}} \sum_{p_{ref} \in \mathcal{P}} \ell(p, p_{ref}) \tag{1}$$

where $\ell(p, p_{ref}) = \max_{t \in \mathcal{T}} \mathbb{1}(p(t) \neq p_{ref}(t))$. This approach can be seen as an instance of CodeT where we don't have to rely on correct test case outputs because they are derived by execution.

2.2 Reflexion

Reflexion (Shinn et al., 2024) is a very recent method that uses linguistic feedback to reinforce language models. In the paradigm of program selection, the idea is that we can improve program generation by augmenting LLM prompts with test cases information. Reflexion can be broken down in the following steps:

1. Prompt the model with a natural language description of a programming problem
2. Generate feedback for the model response in the form of executable test cases
3. Prompt the model again with the original prompt and the test cases feedback

In this way, the model is reinforced with relevant test cases information, guiding it towards more accurate program generation. Reflexion has shown remarkable results on HumanEval and MBPP benchmarks (Zhong et al., 2024; Shinn et al., 2024).

2.3 CodeT

The CodeT algorithm is another method of selecting programs from a set of plausible model generations (Chen et al., 2022). It selects programs based on a dual-execution agreement between the set of generated programs and test cases with the inductive bias that programs with more consistent test cases are better. CodeT works as follows: Based on a natural language input, we sample a set of programs $\mathcal{X}$ and a set of test cases $\mathcal{Y}$ that are consistent with the description. Then, we randomly sample a program-test case pair (x, y) from all possible pairs $\mathcal{D} = \mathcal{X} \times \mathcal{Y}$ until we find a consistent pair.

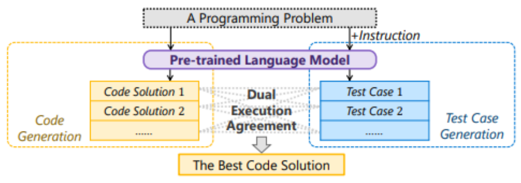


Figure 1: Visualization of the CodeT algorithm. An LLM is used to generate both programs and test cases from a natural language description and the dual-execution agreement algorithm selects the best program.

Next, a set $\mathcal{S}_y$ is generated which consists of all of $y' \in \mathcal{Y}$ that x can pass; similarly, a set $\mathcal{S}_x$ is created which consists of all $x' \in \mathcal{X}$ that pass the same tests as x. We repeat this process k times for different starting (x, y) pairs while keeping track of the generated sets $\mathcal{S}_x$ and $\mathcal{S}_y$ for each pair. Finally, the output of CodeT is a randomly sampled program-test case pair from the set $\mathcal{S}^*{}_x \times \mathcal{S}^*{}_y$ where $\mathcal{S}^*{}_x$ and $\mathcal{S}^*{}_y$ have the largest score, $|\mathcal{S}^*{}_x||\mathcal{S}^*{}_y|$, out of all of the iterations.

As implied by the algorithm, there is a heavy bias towards programs with more test cases. Accordingly, CodeT might encourage poorly trained language models that generate false but consistent test cases. However, in practice, this algorithm is mainly used to improve already state-of-the-art models and thus works well. For example, CodeT improved the pass@1 accuracy of Incoder-6B by over 4% and the pass@10 accuracy over 9% on HumanEval (Fried et al., 2022).

3 Re-Implementation of CodeT

In the original paper, they describe both deterministic and randomized approaches in implementing

CodeT. However, the authors suggest that the randomized implementation, based on the RANSAC algorithm (Fischler and Bolles, 1981), performs better in practice and results in less computational cost. Accordingly, we implemented the randomized CodeT algorithm and describe our results in the following subsections.

3.1 Improvement Over Baseline

To verify that our implementation works as described, we first compare the percent improvements over the baseline (no re-ranking) that CodeT provides. For the most part, the paper uses the Codex (Chen et al., 2021) family of models in their analysis of CodeT. Unfortunately, these set of models are not publicly available anymore. However, they also report their performance on Incoder-6B (Fried et al., 2022), so we report these scores below.

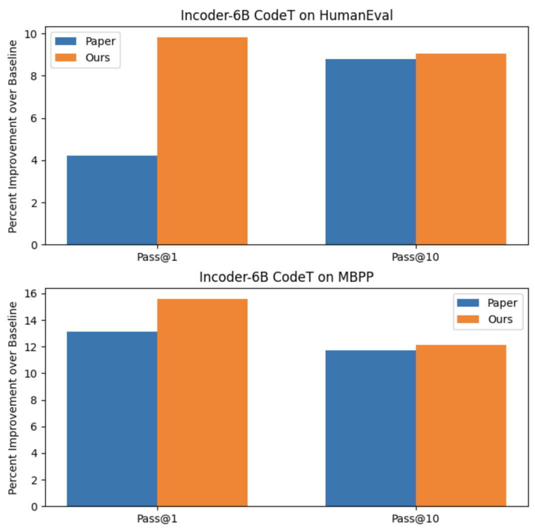


Figure 2: We plot the absolute improvement of CodeT over a no re-ranking baseline from our implementation and the original paper's results on HumanEval and MBPP (the same datasets used in the CodeT paper).

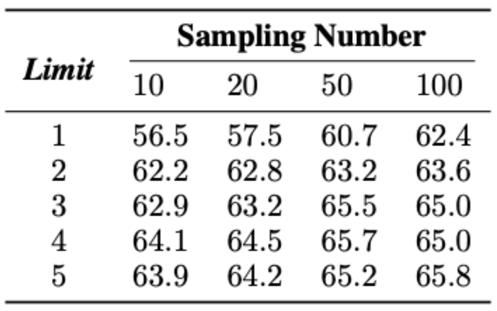
Our implementation does phenomenally well on the pass@1 score and on par for the pass@10 score as shown in Figure 2. We believe that this dramatic increase in performance for the pass@1 score is attributed to our baseline generations being significantly worse than those cited in the paper. For instance, the paper cites 16.4% as the baseline pass@1 score for Incoder-6B, while we could only get a score of 8.8%. This decrease in accuracy leads to significantly more scope for improvement, which our implementation took advantage of. We believe that the authors used an advanced prompting strategy leading to superior results; however, they did not publicly release their prompts, so were unable to recreate the exact baseline score. Yet, the results suggest that our implementation works as expected, since we were able to see a boost in pass@k score after re-ranking.

3.2 Effect on the Number of Tests

The paper also analyzes how the number of tests given to CodeT affects performance. The results indicate that increasing the number of tests improves the pass@1 score of the model. Unfortunately, they report their results on code-davinci-002, one of the closed-source Codex models. However, we perform a similar analysis for our implementation on current state-of-the-art code generation models and report our results in Figures 4 and 5.

	CodeLLaMA	Qwen
HumanEval	18.2	51.1
HumanEval+	14.1	43.2
MBPP	28.3	45.0
MBPP+	22.1	38.2

Table 1: Baseline results for CodeLLaMA-13b and Qwen-2.5-Coder-7b on HumanEval(+) and MBPP(+)



Limit	Sampling Number			
	10	20	50	100
1	56.5	57.5	60.7	62.4
2	62.2	62.8	63.2	63.6
3	62.9	63.2	65.5	65.0
4	64.1	64.5	65.7	65.0
5	63.9	64.2	65.2	65.8

Figure 3: (From CodeT paper) Pass@1 on HumanEval using CodeT with code-davinci-002. Sampling number denotes the number of samples generated by model and Limit denotes the test cases extracted per sample.

In our analysis, we run CodeT for a varying number of input test cases using CodeLLaMA-13b (Roziere et al., 2023) and Qwen-2.5-Coder-7B (Hui et al., 2024). We chose these models in particular because they are two of the best models for code generation and fit reasonably within our computing budget. We also evaluated the algorithm on HumanEval and MBPP as well as their more rigorous counterparts presented by the EvalPlus framework (Liu et al., 2024). Overall, we see very similar

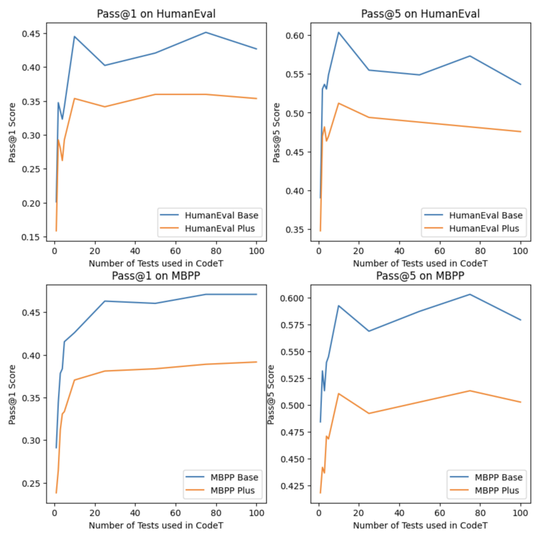


Figure 4: We plot the pass@1 and pass@5 scores of CodeT with CodeLLaMA-13b against the number of input test cases on HumanEval(+) and MBPP(+).

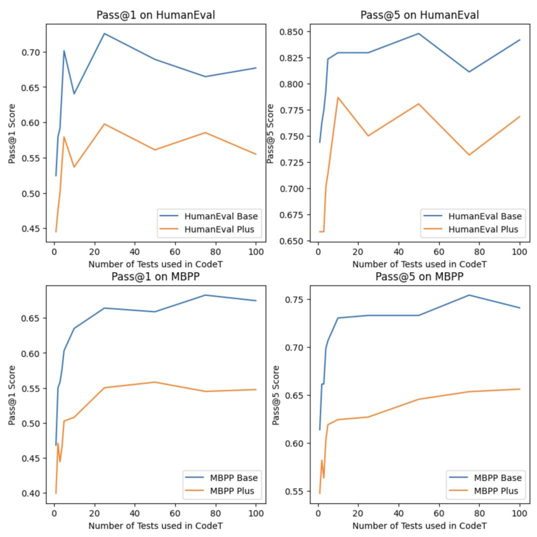


Figure 5: We plot the pass@1 and pass@5 scores of CodeT with Qwen-2.5-Coder-7b against the number of input test cases on HumanEval(+) and MBPP(+).

results to the paper where increasing the number of tests improves performance. The paper reports that after a certain number of test cases, the performance of CodeT can actually decrease because of the existence of spurious test cases (i.e test cases that are correct, but don't have any program filtering capabilities such as assert True). This is shown in our plots where the performance tapers off after a certain number of test cases (and in some cases actually decreases).

4 Augmenting CodeT with Pragmatic Inference

One problem with the CodeT algorithm is that it chooses test cases to filter out programs by randomly sampling from a model. In other words, the algorithm doesn't have a mechanism to select meaningful test cases (i.e edge cases or challenging test cases), which would be useful for filtering spuriously generated solutions. We propose a method to improve upon CodeT by supplementing the algorithm with a procedure to select more *informative* test cases. By modeling test case probability distributions relative to program behavior, we aim to identify and prioritize cases that effectively expose functional discrepancies. This approach is expected to yield improved pass@k metrics while potentially reducing the required test case volume.

4.1 Pragmatic Inference

We aim to use pragmatic (Bayesian) inference to select the best test cases given the program distribution. Say we have the natural language prompts $p \sim P$, the generated programs $c \sim C$ and the generated test cases $t \sim T$; we determine the distribution of test cases given the prompts as follows (see Appendix A for proof):

$$P(t|p) = \sum_{c \in C} P(t|c)P(c|p) \qquad (2)$$

To determine the distribution $P(t|c)$, we can first create a binary "consistency" matrix of the shape (number of tests, number of programs). If a program-test pair is consistent (executes without failure) the corresponding cell in the matrix is 1 (and 0 if not). Then, based on the Rational Speech Acts paradigm of pragmatic inference (Scontras et al., 2021), we can normalize the rows and columns of this matrix, which gives us the distribution of test cases given the programs (Vaduguru et al., 2023).

Now, to determine $P(c|p)$, we can simply use the log-likelihood probabilities given by the model we use (after the softmax of the logits). Using these two probability distributions and the given equation, we can determine the conditional distribution of test cases over the natural language inputs and rank them based on their likelihood.

5 Methods

The overall pipeline of our project is as follows:

1. Sample n programs and test cases from a language model

2. Create the consistency matrix indicating which program-test pairs are consistent
3. Determine the most informative tests given the programs using pragmatic inference

We describe each of these components in more detail in this section.

5.1 Sampling Code from LLMs

For each programming task, we used vLLM (Kwon et al., 2023) to sample 100 programs and test cases per model. We believed that 100 generations for each would be large enough to achieve meaningful results with RSA, while staying within the computing budget for our project. In terms of prompting the language models, we stuck to code-completion models (non-instruction-tuned) as they are easier to prompt and don't require any elaborate prompting strategy. Our prompting strategy and hyperparameters are detailed further in Appendix C.

5.2 Creating the Consistency Matrix

We then created a consistency matrix M for each set of generations by executing program j on test case i and setting $M[i][j] = 1$ if the execution was successful. Our programs were generated as Python functions and our test cases were simple assert statements. So, to determine whether the program-test pair (c, t) was consistent, we simply executed the code string $c + \backslash n + t$ using Python's "exec" function.

5.3 Pragmatically Choosing Test Cases

To choose test cases informatively, we first need to find each probability value from equation 2. To generate the test case distribution, we sampled the log-likelihood ($\log P(c|p)$) of the program from the language model. To generate the conditional distribution of the test cases given the programs ($P(t|c)$), we applied RSA to the consistency matrix. We first generate the distribution $P_{L_0}(c|t)$ (the probability distribution of programs given test cases of an un-informed model) by normalizing over the rows of the consistency matrix. Then we normalize over the columns of this matrix to create the distribution $P_{S_1}(t|c)$, representing the probability of a test case given a program of a pragmatic language model. With the log-likelihood probabilities from the model and conditional test case distribution from the RSA matrix, we then use the formula described in equation 2 to generate the overall test case distribution.

6 Results

In this section, we present the results we achieved with our augmented version of CodeT against the baseline implementation.

6.1 Running CodeT with Pragmatic Test Cases

In our first experiment, we decided to test how CodeT performs when we use random test cases from our model generations vs pragmatically selected test cases using our method.

	CodeT	CodeT + RSA
HumanEval	20.1	**25.2**
HumanEval+	34.8	**41.0**
MBPP	30.3	**35.0**
MBPP+	25.1	**31.3**

Table 2: Pass@1 score when we use a single test case to filter out programs using CodeT.

In our results, we observed that our method performs strongly when we use a single test case in CodeT. However, for using a larger number of test cases in CodeT, we found that our method performs very similarly to CodeT without our augmentation. This indicates that selecting test cases pragmatically is useful when there aren't many test cases to use for filtering; however, using more test cases for filtering programs is a better option than using a single informative test case.

6.2 Pragmatic Test Selection with Ground Truth Programs

When we observed the test cases that were being selected by our algorithm, we realized a major issue: spurious programs with higher likelihoods were causing incorrect test cases to be selected in our algorithm. Essentially, since we didn't use the ground-truth program in our test selection, incorrect test cases were being used in CodeT, resulting in poor program selection. Hence, we decided to retry pragmatic test case selection with RSA using the ground-truth program. Instead of weighting each model-generated program by its likelihood from the model, we just used the test case distribution conditioned on the ground truth program. Equation 2 can then be written as follows (proved in Appendix B)

$$P(t|p) = P(t|c_{\text{true}}) \tag{3}$$

The procedure for generating this distribution is identical to that in the previous experiment; however, we just take the test cases from the column with the ground-truth program, which eliminates the need to weight generated programs by their likelihood.

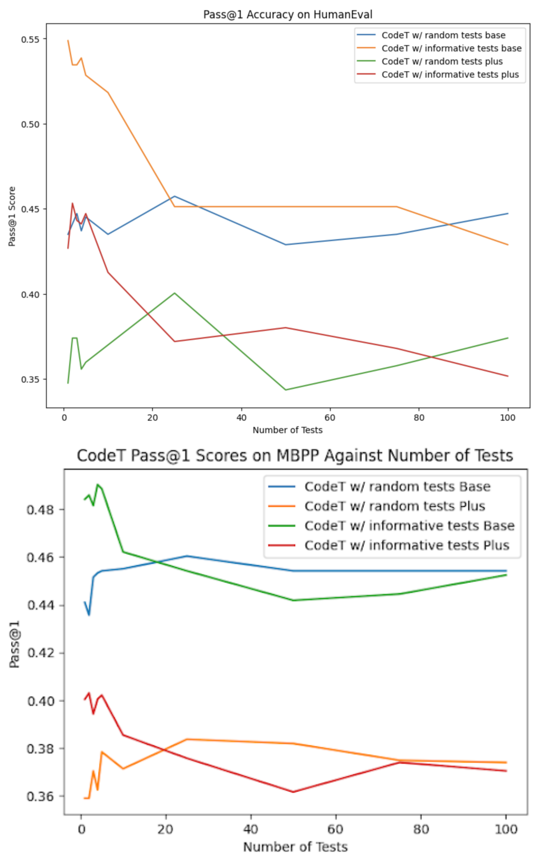


Figure 6: We plot the pass@1 scores of CodeT with pragmatic test cases using the ground truth program vs randomly chosen test cases.

These results indicate that using information about the ground-truth program helps our algorithm select better test cases for CodeT. We see a massive improvement in Pass@1 accuracy with informative test cases compared to vanilla CodeT. Much like our results without the ground-truth program, we only see a significant difference in performance when the number of test cases are fewer. However, using the ground-truth program shows performance gains for up to 10 test cases used in CodeT, which is a more significant gain than just a single test case as we saw previously.

6.3 Visualizing Pragmatic vs Random Test Cases

We also directly viewed the test cases that were being selected by our algorithm, to subjectively decide whether the the pragmatic test cases were "better" than randomly chosen test cases. Some of the examples we found are as follows:

1. is_prime: the problem is to generate a program that determines whether a number is prime or not. Our algorithm selects the test case `assert isPrime(1) == False`, which is an edge case many programs miss, indicating that our algorithm correctly identifies the tests that correct programs must pass.

2. has_close_elements: the problem is to determine whether two elements in the array are within some threshold of each other (exclusive). Our algorithm selects the test cases `assert has_close_elements([1, 2, 3], 1) == False`, which is an edge case for elements being *strictly* within the threshold of each other. Many programs fail this test because they don't require elements to be strictly greater or less than each other.

3. remove_duplicates: the problem is to remove any elements in the array that occur more than once. Our algorithm chooses the test case `assert remove_duplicates([1, 1, 1, 1]) == []`, which checks whether the program correctly removes all elements in the array since they are identical. Many incorrect programs fail when the output is an empty list.

7 Future Work

The results of our project motivate several areas of future work that we are interested in pursuing. We highlight some of these areas below:

7.1 Using other Execution Agents

In our implementation, we focused on augmenting CodeT with our method of selecting test cases. It is a natural extension of our work to experiment with other execution-based program selection methods like MBR-Exec.

7.2 Using Pragmatic Inference to Guide Fine-tuning

In our experiments with using ground-truth programs to guide pragmatic inference for test case selection, we observed that program selection improved tremendously with CodeT. We also saw that this approach frequently chose edge cases that highlighted key properties of correct programs. An interesting idea would be to use this strategy of selecting test cases to fine-tune a test generation model. Instead of fine-tuning on a random set of consistent test cases, we would be fine-tuning on these *informative* test cases to improve test case synthesis through supervised fine-tuning.

7.3 Using Pragmatic Test Cases in Reflexion

Reflexion is a recently championed approach in improving program synthesis by providing test cases in an inference-time reinforcement learning paradigm. In the original paper, the authors provide randomly chosen consistent test cases while prompting an LLM to improve its ability to generate correct programs. With our work on pragmatic test cases, it would be interesting to see if we can improve upon this by using the pragmatically chosen test cases via RSA. Since our method chooses edge cases with higher probability, we expect that the reflexion-augmented model would generate correct programs with higher probability.

8 Conclusion

In this paper, we propose a novel method of augmenting execution-based program selection using pragmatics. We start by presenting the current state-of-the-art techniques in program selection from language model generations including CodeT, MBR-Exec and Reflexion. We then re-implement the most widely used technique, CodeT, and achieve similar results to that of the original paper. In doing so, we identified a key flaw in the CodeT method, namely that it has no method of choosing informative test cases to filter out unwanted programs. In our approach, we tackle this challenge by using pragmatic inference to select better test cases to improve the selection algorithm. We found that we were able to improve the pass@k scores when the number of available test cases was limited. Our results imply that there is tremendous scope for pragmatics in program selection and we hope to explore such areas in future work.

References

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*.

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

Martin A. Fischler and Robert C. Bolles. 1981. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM*, 24(6):381–395.

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

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

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626.

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

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36.

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

Gregory Scontras, Michael Henry Tessler, and Michael Franke. 2021. A practical introduction to the rational speech act modeling framework. *arXiv preprint arXiv:2105.09867*.

Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I Wang. 2022. Natural language to code translation with execution. *arXiv preprint arXiv:2204.11454*.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2024. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36.

Saujas Vaduguru, Daniel Fried, and Yewen Pu. 2023. Generating pragmatic examples to train neural program synthesizers. *arXiv preprint arXiv:2311.05740*.

Li Zhong, Zilong Wang, and Jingbo Shang. 2024. Ldb: A large language model debugger via verifying runtime execution step-by-step. *arXiv preprint arXiv:2402.16906*.

A Proof of Equation 2

Proof.

$$\begin{aligned} P(t|p) &= \sum_{c \in C} P(t, c|p) \\ &= \sum_{c \in C} P(t|c, p) P(c|p) \end{aligned}$$

But, I argue that $P(t|c,p) = P(t|c)$ because test case generation is conditionally independent of the prompt given the code for the program it is testing. If a model is given the entire code for a particular program, it does not need the original prompt as well to generate test cases. Hence,

$$P(t|p) = \sum_{c \in C} P(t|c) P(c|p)$$

This concludes the proof. □

B Proof of Equation 3

Proof. From the previous section, we have that

$$P(t|p) = \sum_{c \in C} P(t|c) P(c|p)$$

Now, if we select test cases based on the ground-truth program, we have

$$P(t|p) = P(t|c_{true}) P(c_{true}|p)$$

But, $P(c_{true}|p) = 1$ because the ground-truth program is correct by definition. Hence,

$$P(t|p) = P(t|c_{true})$$

This concludes the proof. □

C Prompting Code Completion Models

In this section, we present our prompt templates for generating programs and test cases from the code completion models we used in our experiments.

C.1 Prompts for Program Synthesis

```
PROGRAM_PROMPT = """
# Complete the following function
    definition.
{<function header>}
"""
```

C.2 Prompts for Test Synthesis

```
TEST_PROMPT = """
# Write test cases as assert
   statements for the following
   function header.
def <some_function>:
    """
    Some docstring describing the
        function.
    """
assert
"""
```

Prompting models for test case synthesis involved a few additional steps compared to program synthesis. First, we made sure that the model outputs tests as assert statements as opposed to Python's unit-test format because the model produced more accurate tests in this setting. Additionally, we removed all of the test cases from the description of the function in the docstring to encourage the model to produce more diverse test cases.

C.3 Generation Hyperparameters

We used the following hyperparameters during program and test case synthesis:

- Temperature: 0.8
- Max Tokens: 128
- Num Generations: 100
- Top P: 0.95

We decided to use a high temperature to promote the generation of diverse programs and test cases since we were sampling 100 generations per run. We also set the max tokens to 128 to prevent the model from generating extra irrelevant code. Finally, we used top p sampling to add to the generation diversity and ensure the programs had reasonably high likelihood.