scale

Planning In Natural Language Improves LLM Search For Code Generation

Evan Wang[1,2], **Federico Cassano**[°3,4], **Catherine Wu**[°], **Yunfeng Bai**[1], **Will Song**[1], **Vaskar Nath**[1], **Ziwen Han**[1], **Sean Hendryx**[1], **Summer Yue**[1], **Hugh Zhang**[1]

[1]Scale AI, [2]California Institute of Technology, [3]Northeastern University, [4]Cursor AI,
[°]*Work conducted while at Scale AI*

Correspondence to evan.wang@scale.com and hugh.zhang@scale.com

Abstract

While scaling training compute has led to remarkable improvements in large language models (LLMs), scaling inference compute has not yet yielded analogous gains. We hypothesize that a core missing component is a lack of diverse LLM outputs, leading to inefficient search due to models repeatedly sampling highly similar, yet incorrect generations. We empirically demonstrate that this lack of diversity can be mitigated by searching over candidate plans for solving a problem in natural language. Based on this insight, we propose PLANSEARCH, a novel search algorithm which shows strong results across HumanEval+, MBPP+, and LiveCodeBench (a contamination-free benchmark for competitive coding). PLANSEARCH generates a diverse set of observations about the problem and then uses these observations to construct plans for solving the problem. By searching over plans in natural language rather than directly over code solutions, PLANSEARCH explores a significantly more diverse range of potential solutions compared to baseline search methods. Using PLANSEARCH on top of Claude 3.5 Sonnet achieves a state-of-the-art pass@200 of 77.0% on LiveCodeBench, outperforming both the best score achieved without search (pass@1 = 41.4%) and using standard repeated sampling (pass@200 = 60.6%). Finally, we show that, across all models, search algorithms, and benchmarks analyzed, we can accurately predict performance gains due to search as a direct function of the diversity over generated ideas.

"If you fail to plan, you plan to fail." — Mastermind, Taylor Swift

1. Introduction

The bitter lesson [40] famously posits that two forms of scaling trump everything else: learning and search. While recent advances in large language models have removed all doubt on the effectiveness of learning, search has not yet proven its value for large language models, despite its success with classical machine learning techniques [4, 7, 8, 10, 17, 37, 38].

Here, we refer to search as any method of spending compute at inference time to improve overall performance [28]. In this work, we focus our efforts on improving LLM search for code generation, one of the most important current applications of LLMs. We hypothesize the major bottleneck preventing widespread use of search at inference time for code is a lack of high-level diversity in model outputs. This lack of diversity is likely in part due to specific post-training objectives commonly used to train

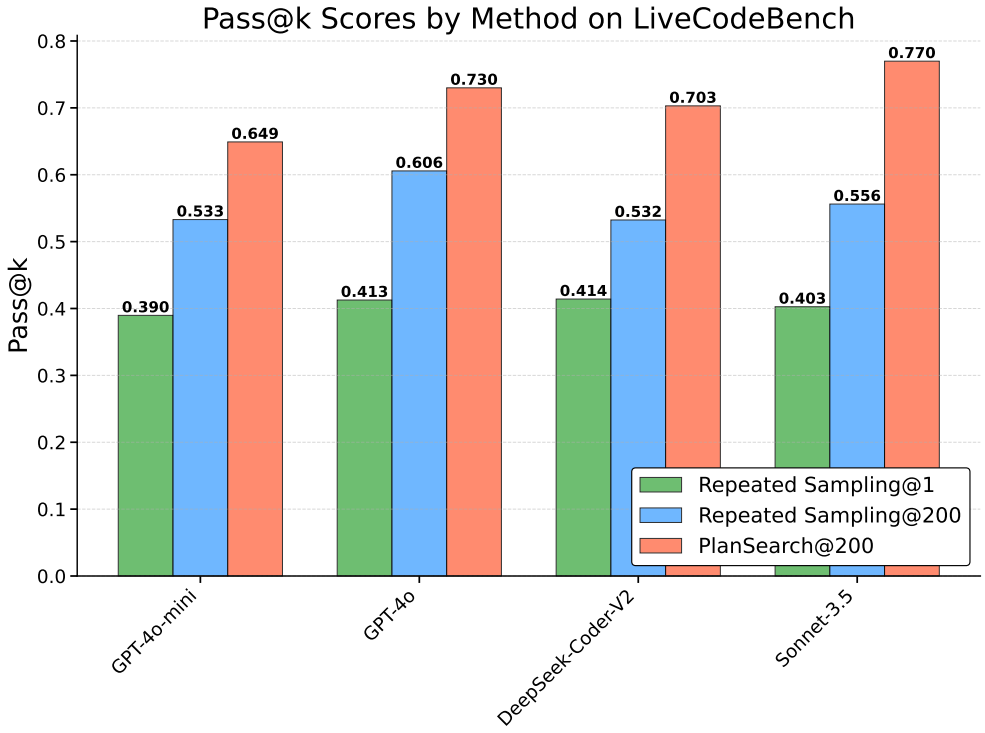


Figure 1: Comparison of REPEATED SAMPLING, both pass@1 and pass@k, and our novel method PLANSEARCH. On every model, our method outperforms baselines by a wide margin, with the best model-method combination of Claude 3.5 Sonnet / PLANSEARCH achieving performance nearly double that of the best model without search.

LLMs as chatbots, in which models are oftentimes optimized to produce a single correct answer [31, 33]. We empirically demonstrate that this is the case for many open-source language models which have undergone significant post-training. Specifically, we show that in many cases, despite instruction tuned models outperforming base models by large margins on a single sample regime (pass@1), this trend disappears—sometimes even reversing—when generating many samples. We refer to Figure 3 as an example of this phenomenon.

Furthermore, the lack of diversity is particularly harmful for search algorithms. In the most egregious of cases with little to no diversity, such as greedy decoding, repeated sampling from the model returns highly similar programs, resulting in minimal gain from additional inference-time compute. This diversity problem is also not reflected in many public leaderboards (e.g. LMSYS Chatbot Arena [14], LiveCodeBench [22], OpenLLMLeaderboard [1]), which often report only the pass rate from a single sample of the model, ignoring an entire dimension along which to compare models. While the performance of one sample is the primary metric of relevance for applications such as chatbots, as users typically are sensitive to latency, this single scalar is insufficient to fully capture the quality of a model when it is allowed to use more inference time compute.

In this paper, we explore several directions for improving the diversity of LLMs at inference time. We hypothesize that the right axis of diversity to search over is the natural language conceptual/idea space, and we validate our hypothesis across several experiments. First, we show that models can produce the correct final program when fed the correct solution sketches, where these sketches have been "backtranslated" from passing solution code into sketches in idea space (Section 3.2). Second, we show that when models are asked to generate their own ideas before implementing them on LiveCodeBench (IDEASEARCH), their accuracy conditioned on a particular sketch trends towards either 0% or 100%,

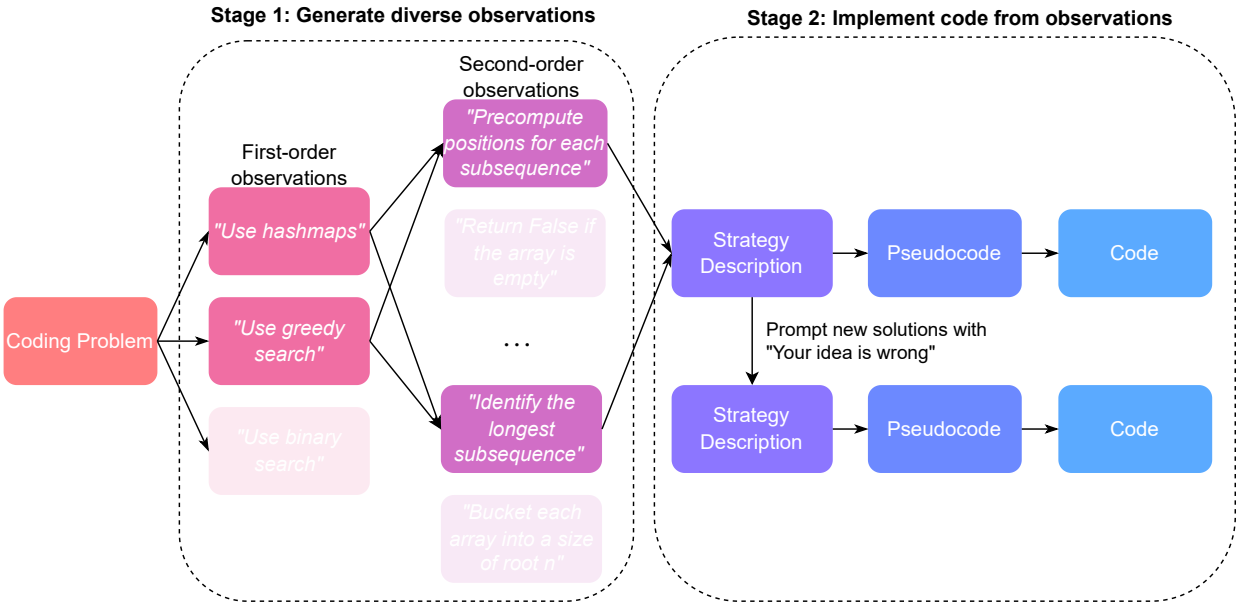


Figure 2: An example trajectory of PLANSEARCH, which searches over plans in natural language as a method of increasing diversity in the search process. PLANSEARCH first generates observations, then combinatorially samples subsets of these observations to generate the next step in the search process. To generate the next layer of observations, the combinations derived from the first observations are used as a stepping stone to generate the next observations, and the process repeats. After generating both the first and second order observations, PLANSEARCH then generates a natural language description of a strategy to solve the problem. For additional diversity, the model is prompted to regenerate its strategy as an additional sample before generating code. See Section 4.2 for additional discussion.

suggesting that most of the variance in passing a particular problem is captured by whether the sketch is correct rather than any other factor. These two experiments suggest a natural method to improving LLM search for code generation: by searching for the correct idea to implement.

Guided by this principle of *maximizing exploration of ideas*, we propose PLANSEARCH. In contrast to many existing search methods that search over individual tokens [44, 46], lines of code [24], or even entire programs [25], PLANSEARCH searches over possible *plans* for solving the problem at hand, where a plan is defined as a collection of high level observations and sketches helpful to solve a particular problem (Figure 2). To generate novel plans, PLANSEARCH generates a number of observations about the problem, before combining these observations into a candidate plan for solving the problem. This is done for every possible subset of the generated observations to maximally encourage exploration in idea space, before the codes are eventually all translated into a final code solution (Section 4.2). We find that searching over plans outperforms both standard repeated sampling and directly searching over ideas (IDEASEARCH, introduced in Section 4.1.2) in terms of effectively using compute at inference time.

Applying PLANSEARCH on top of Claude 3.5 Sonnet achieves a state-of-the-art pass@200 of 77.0% on LiveCodeBench, outperforming both the best score achieved without search (pass@1 = 41.4%) and the standard best-of-n sampling score (pass@200 = 60.6%). Furthermore, consistent with recent findings on the effectiveness of search on top of small models [5, 6, 12, 42], allowing PLANSEARCH based on a small model (GPT-4o-mini) outperforms larger models not augmented with search after merely 4 attempts. Evaluations of PLANSEARCH across two other coding benchmarks, HumanEval+ and MBPP+ [26], suggest similar improvements.

Finally, we measure the diversity of output code over the idea space of all search methods via an LLM-as-a-judge procedure (Section 6.1) and show that the resulting diversity score is highly correlated with the performance gains generated by that search method. This provides further support for our hypothesis that the effective exploration of plans in idea space is key to LLM search for code generation (Figure 6).

2. Related Work

We reiterate that search as defined in the context of our paper refers to any method which expends inference time compute to improve performance. We further specify planning as any form of high level observation or abstract thought that assists a model in generating a final solution. Our work builds off a long history of work in scaling search and planning.

2.1 Search in Classical AI

Classical search algorithms like breadth-first search, depth-first search, and A* search have been widely used for pathfinding, planning, and optimization [34]. More advanced search techniques like Monte Carlo tree search (MCTS) have achieved remarkable success in domains like game playing, enabling superhuman performance in Go [37, 38], poker [7, 8] and Diplomacy [17]. More recently, Jones [23] find scaling laws for the performance of AI systems in board games, where ELO improves logarithmically with the amount of compute spent at inference.

2.2 Search with Language Models

Applying search on top of LLMs has been a topic of much interest, especially with an eye towards code generation [13, 25]. Historically, methods such as beam search significantly improved performance for translation systems [18]. Closer to the present day, several recent works have explored repeated sampling [5, 6, 12, 42] as a search method for improving performance. Repeated sampling is a method which directly generates candidate code solutions from the model many times at moderate to high temperatures in hopes that one of the resulting generations will be correct. However, although these works address the roughly linear increase in pass@k with respect to $\log k$, they only focus on the most basic version of repeated sampling, without searching in idea space.

When combined with a verifier, reward model, or other filtering algorithm to select the best generation (in cases where pass@k is not a viable metric due to lack of test cases), it is also known under the name of best-of-n sampling [29]. Many works show somewhat good results under intelligent selection of such a filtering algorithm [11, 12]. Recently, several approaches have demonstrated the power of repeated sampling. For example, repeated sampling from a small model can sometimes outperform taking a single sample from a large model on an equalized compute bases [39]. Unlike algorithms such as repeated sampling, which search over the output space, the key insight of PLANSEARCH is that it is far more effective to instead search plans over the *latent idea space*. By explicitly searching over different natural language plans before generating the code, we significantly increase the diversity of the final code outputs and thus, the resulting pass@k scores for sufficiently large k.

Regarding searching over plans in natural language, several approaches have also proposed generalizing chain-of-thought [41] reasoning into a search-like process, such as Tree of Thoughts [43] and Reasoning via Planning [20]. However, prior methods have largely demonstrated effectiveness on somewhat contrived problems designed to highlight the power of search, such as the game of 24, or classic planning benchmarks such as Blocksworld [27], where both benchmarks are easier to solve by explicitly considering many options, and where the 'steps' over which to search over are fairly obvious. By contrast, most real-world planning is used to assist in domains that are complex enough to benefit from, but

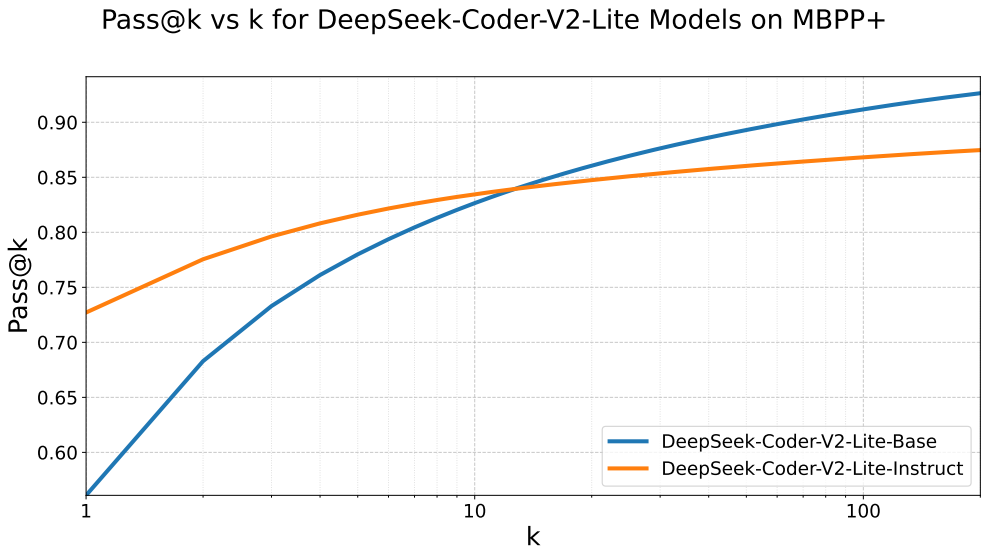


Figure 3: Despite DeepSeek-Coder-V2-Lite-Base having significantly lower pass@1 than its instruct counterpart, we observe that this trend reverses as k increases, suggesting that the instruct model has less diversity than its base model counterpart. We observe this trend for many, but not all, models and benchmarks, and provide the full data in Appendix H.

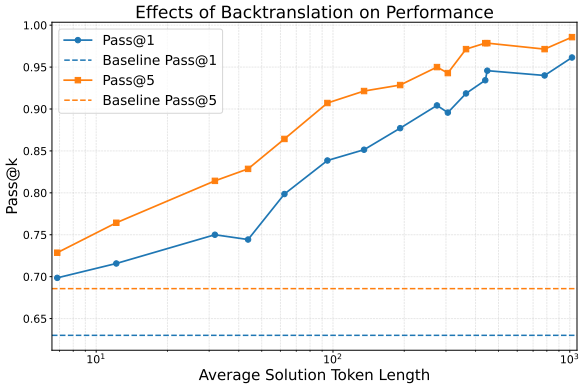
not require, the additional exploration of plans. We demonstrate that PLANSEARCH, which plans in natural language, outperforms baseline search methods in one such domain: code generation. Moreover, our analysis reveals the underlying reason that such search is effective: it increases the diversity of the generated ideas, allowing more efficient search relative to other methods which repeatedly submit highly similar, incorrect solutions.

3. Motivation

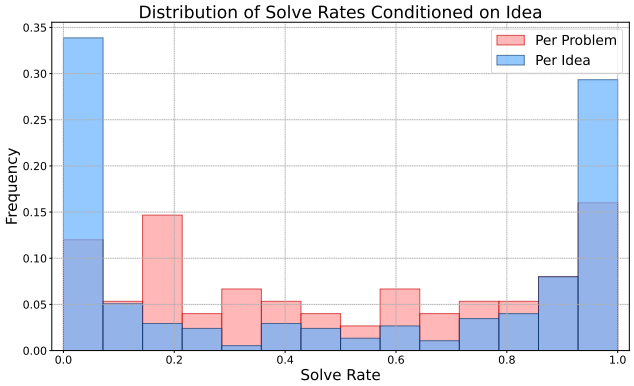
Coding is a powerful area in which search should excel. While search in other domains requires both generating many solutions *and* selecting the correct solution amongst all the resulting generations, coding often only requires the former, as any valid piece of code can be tested via code execution against given test cases. This allows code search algorithms to sidestep many of the issues that plague search algorithms for more open-ended domains (e.g. generating poetry) due to difficulty in selecting correct solutions out of all the generated solutions.

3.1 Defining the Search Space

Perhaps the most important question for eliciting strong search capacities is determining which space to search over, as finding the proper layer of abstraction is critical to progress in the field. Prior approaches have varied, with many people searching over individual tokens [44, 46], lines of code [24], or even entire programs [25]. We hypothesize that the key factor is obtaining *the correct solution sketch*, which we define as a description of the correct program in natural language space. Intuitively, conducting the reasoning process in natural language space allows us to effectively harness the training process of LLMs, which have observed many human reasoning traces in both pre- and post-training. Prior work [41] has observed strong positive effects from being allowed to conduct such reasoning in natural language, making it a natural place to search over. We describe two experiments providing evidence for this hypothesis by testing on the LiveCodeBench benchmark using GPT-4o-mini as our model.



(a) Performance of GPT-4o-mini on LiveCodeBench when provided with backtranslated solutions of varying lengths. The baselines plot performance without backtranslated solutions. Providing the model with a compressed solution in natural language, even as short as 10 tokens, significantly increases performance.



(b) We plot the distribution of solve rates conditioned on being given a solution sketch and without. When conditioning on a given sketch, we notice that downstream solve rates polarize towards either 0% or 100%. Most of the variance in performance is predicted by whether a given idea is correct or not.

Figure 4: Backtranslation shows the promise of providing good sketches, and conditioning on idea shows the presence of a solution sketch polarizes performance.

3.2 Backtranslation

To investigate the hypothesis whether the idea space, instantiated as solution sketches, is the right area of exploration, a natural question is whether LLMs can correctly implement a correct code solution given a correct sketch. Inspired by approaches to backtranslation in machine learning [16, 32, 35], we experiment with "backtranslating" passing code solutions back into idea space. First, we generate code solutions using GPT-4o to generate 1000 attempts to solve the problem and filter out problems without any passing solutions. As we also do not have a dataset of correct solution sketches associated with each solution, we generate a candidate correct idea via backtranslation. We do this by feeding an LLM both the problem and code solution and asking the LLM to convert said solution into a natural language description of the solution. Additionally, we vary the detail of the backtranslated idea via instructions to the LLM in the prompt (e.g. 'in w words'). A full description of the prompts can be found in Appendix J.1, alongside several example backtranslated solutions of various lengths.

We observe that prompting a model with a backtranslated idea significantly improves accuracy, increasing with the length of the translated idea (Figure 4a), which suggests that having a correct sketch is sufficient to produce the correct final solution with relatively high accuracy, even only after 10 tokens of backtranslated solution. This suggests that the correct direction of search is to explore through idea space to maximize the chance of arriving at a correct idea.

3.3 Conditioning on Idea Quality

In a follow-up experiment, we prompt an LLM to generate its own sketches to solve LiveCodeBench problems instead of providing it with golden ones via backtranslation. First, we generate 5 ideas per problem using IdeaSearch, defined in Section 4.1.2. For each idea, we then sample 25 candidate solutions and measure their pass rate. For this experiment, we filter out any problem that GPT-4o-mini solves with either a 100% or a 0% solve rate, since such problems are either too easy or too hard for the model and would not be informative for this experiment. We end with 75 problems and 375 sketches.

To test our hypothesis that generating a correct sketch is a critical factor for solving problems, we compare the distribution of solve rates for generating correct code solutions *conditioned* on a given sketch to the distribution over solve rates given a sketch drawn at random, i.e., just the distribution over solve rates.

Formally, for any problem P_i, we sample some sketch I from some conditional distribution with probability mass $P(I|P_i)$. The probability of solving P_i is then $P(\text{solve}|P_i, I)$. We compare the solve-rate distribution, $P(\text{solve}|P_i, I)$ over all problems and all sketches *versus* the solve-rate distribution of $\sum_I P(\text{solve}|P_i, I) \cdot P(I|P_i) = P(\text{solve}|P_i)$ over all problems.

While verifying whether a sketch is correct or incorrect is difficult without access to external labels, a key insight is that if generating the correct idea is a critical factor in solving the problem, then conditioning on a particular sketch should polarize the distribution of solve rates towards $\{0, 1\}$. If the model is given a correct sketch, it should consistently generate correct solutions, while if given a bad sketch, it should consistently generate incorrect solutions.

Our results confirm this to be the case. Figure 4b shows the distribution of solve rates across problems, both unconditionally (in red) and conditioned on each sketch (in blue). We notice that when grouping by sketches, the solve rates indeed become polarized towards $\{0, 1\}$. This result has important implications for improving code generation, suggesting that a large portion of variance in performance may come from whether the model is able to generate a correct idea or not. Therefore, a natural path for improvement is to focus on the sketch generation step and search for correct sketches and observations in idea space before generating solution code.

4. Methods

We provide a description of the various methods of search we explore in our work. If additional background on competitive programming and related notation is desired, we provide more (optional) information in Appendix K.

4.1 Baselines

4.1.1 Repeated Sampling

We consider the basic prompting approach as a baseline, in which we use few-shot prompting by providing the LLM with a number of problem-solution pairs before asking it to solve the desired question [9]. A full example of the prompt is given in Appendix J.2. In code generation, the most common variant of search utilized is repeated sampling, where models are repeatedly sampled from until they generate an output that passes the test or the maximum number of samples is reached. Refer to the Related Work for more information (Section 2.2).

4.1.2 IdeaSearch

A natural extension of the Repeated Sampling approach discussed in Section 4.1.1 is to avoid prompting the LLM for the solution code immediately. This can be viewed as an application of the commonly used "chain-of-thought" prompting to programming problems [41], although we find that IdeaSearch shows non-negligible performance boosts over standard "chain-of-thought" prompting (see Appendix E).

In IdeaSearch, the LLM is given the problem P and is asked to output a natural language solution S of the problem. Then, a separate instance of the LLM is given P and S, and tasked to follow the proposed solution S to solve the problem P. The purpose of IdeaSearch is to isolate the effectiveness of having the correct "idea/sketch" for solving the problem. Empirically, we find that explicitly forcing the search

algorithm to articulate an idea for solving the problem increases diversity. See Appendix J.3 for detailed prompts.

4.2 PLANSEARCH

While both REPEATED SAMPLING and IDEASEARCH are successful and lead to improvement in the results on benchmark results, we observe that in many of the cases, prompting multiple times (pass@k) (even at high temperatures) will only lead to small, narrow changes in the output code that change minor aspects but fail to improve upon pitfalls in idea.

4.2.1 Prompting for Observations

Starting from the problem statement P, we prompt an LLM for "observations"/hints to the problem.

We denote these observations as O_i^1, where, $i \in \{1, \ldots, n_1\}$ due to the fact that they are first-order observations. Typically, n_1 is on the order of 3 to 6. The exact number depends on the LLM output. To use these observations to inspire future idea generation, we create all subsets with size at most 2 of $S^1 = \{O_1^1, \ldots, O_{n_1}^1\}$. Each of these subsets is a combination of observations, and for clarity we denote each subset as $C_i^1, i \in \{1, \ldots, l_1\}$, where $l_1 = 1 + n_1 + \binom{n_1}{2}$.

4.2.2 Deriving New Observations

The set of all observations can be thus defined as a directed tree with depth 1, where the root node is P, and an edge exists for each C_i^1 pointing from P to C_i^1. We then repeat this procedure from Section 4.2.1 on each leaf node C_i^1 to generate a set of second order observations, $S_i^2 = \{O_{i,1}^2, \ldots, O_{i,n_{i,2}}^2\}$. To obtain second order observations, we prompt the model with both the original problem P and all observations contained in C_i^1, framed as primitive observations that are necessary in order to solve P. The LLM is then prompted to use/merge the observations found in C_i^1 in order to derive new ones.

The same procedure as Section 4.2.1 is used to create all subsets $C_{i,j}^2$, for all $i \in \{1, \ldots, l_1\}$. This process may be arbitrarily repeated, but we truncate the tree at depth 2 for computational constraints.

Note that there is no assumption any of the observations generated are correct. In fact, it is critical to note that many of them may be incorrect. The observations merely serve to elicit the model to search over a more diverse set of ideas.

4.2.3 Observations to Code

After the observations have been made, they must be implemented as ideas before being translated into code. For each leaf node, we prompt the model with all observations, along with the original problem P, in order to generate a natural language solution to the problem P. To add more diversity, for each generated idea, we generate an additional idea by supposing the idea is wrong, and asking an LLM to give criticisms/feedback, thus increasing our proposed ideas by a factor of 2.

These natural language solutions are then translated into pseudocode, which are subsequently translated into actual Python code. We take a more granular approach to reduce the translation error (which may cause the model to revert to its original mode, disregarding the reasoned-through observations). We provide all prompts for all sections in Appendix J.4.

Model	Benchmark	Pass@1	Pass@200	IDEASEARCH@200	PLANSEARCH@200
GPT-4o-mini	LiveCodeBench	39.0	53.3	59.4	64.9
GPT-4o	LiveCodeBench	41.3	60.6	70.4	73.0
DeepSeek-Coder-V2	LiveCodeBench	41.4	53.2	65.9	70.3
Claude-Sonnet-3.5	LiveCodeBench	40.3	55.6	70.2	77.0
GPT-4o-mini	HumanEval+	83.7	95.0	97.5	98.2
GPT-4o	HumanEval+	86.4	98.2	97.6	99.5
DeepSeek-Coder-V2	HumanEval+	82.8	91.4	97.2	99.3
Claude-Sonnet-3.5	HumanEval+	81.6	88.9	95.6	98.5
GPT-4o-mini	MBPP+	73.5	83.8	87.3	91.0
GPT-4o	MBPP+	77.2	87.4	89.3	92.2
DeepSeek-Coder-V2	MBPP+	76.3	81.9	89.1	92.6
Claude-Sonnet-3.5	MBPP+	77.1	83.0	87.8	93.7

Table 1: We find that PLANSEARCH and IDEASEARCH improve upon search baselines across all models, with PLANSEARCH achieving the best results across all models and benchmarks considered. Notably, using PLANSEARCH on top of Claude 3.5 Sonnet [2] has a pass@200 of 77.0 on LiveCodeBench, which is nearly double the performance of the top model without using search (41.4). We highly encourage readers to check Appendix A for complete results and pass@k curves.

5. Experimental Results

5.1 Datasets

We evaluate our search methods on three benchmarks: MBPP+, HumanEval+ [26], and LiveCodeBench [22]. MBPP [3] and HumanEval [13] are some of the most widely used code benchmarks in the field. However, since both benchmarks provide only a few test cases, [26] updates both benchmarks with additional test cases that increase the benchmarks' robustness to reward hacking. LiveCodeBench is a benchmark for coding that consists of competitive programming problems which typically require advanced reasoning capabilities. Given the reality that coding data is often highly upsampled during pre-training [15, 30], LiveCodeBench differentiates itself from other benchmarks by taking care to segregate problems by date to avoid data contamination concerns. For this paper, we use only the subset of problems between May 2024 and September 2024 to avoid possibilities of contamination. We choose May 2024 as the cutoff date to ensure that our results with our best performing model (Claude 3.5 Sonnet) are not due to contamination, because Claude 3.5 Sonnet has a knowledge cutoff of April 2024. To ensure fair comparison, we use the same cutoff for all models evaluated, even though the precise cutoff dates for other models may vary slightly from May 2024.

5.2 Experiment Details

For all search algorithms, we require that all output code be in the correct format specified, and we mark a solution as incorrect if it does not follow the intended formatting. The extracted code is then run through all tests of the program and marked as correct if and only if it passes all tests.

All models are run with temperature 0.9 and top-p of 0.95. Temperature was determined through a coarse hyperparameter sweep on REPEATED SAMPLING and IDEASEARCH from $T \in \{0.0, 0.1, 0.2, \ldots, 1.2\}$, which we describe in Appendix F.

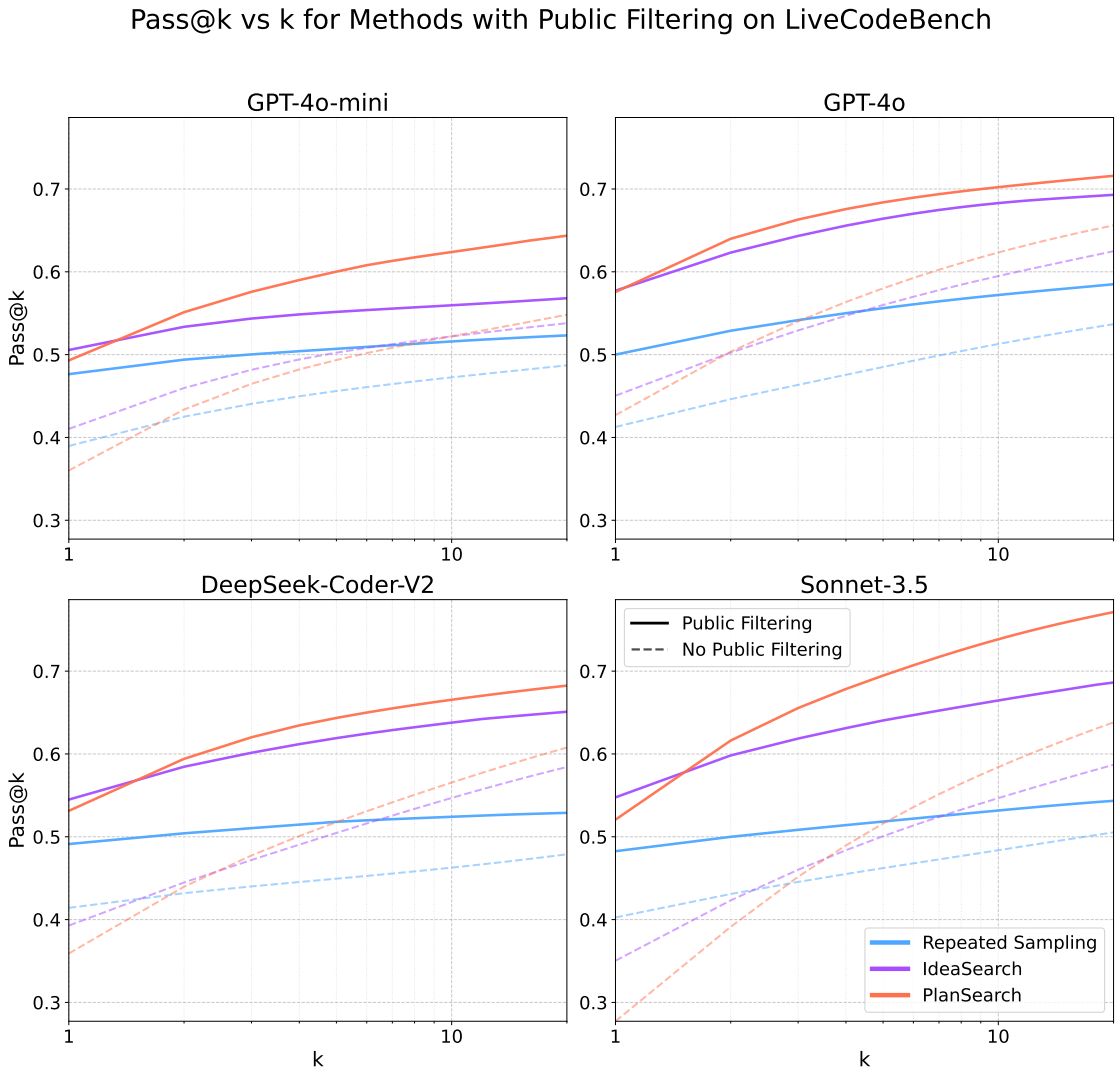


Figure 5: Performance of all models and methods on LiveCodeBench with public test filtering. The idea of public test filtering is to shift pass@k curves leftward (i.e., bringing high k to low k), so we plot curves in detail over $k \in \{1, \ldots, 20\}$. Dotted lines are provided for reference of the base method pass@k before filtering. Even at 10 completions, PLANSEARCH outperforms *filtered* REPEATED SAMPLING by a flat 30 to 40%. Again, full pass@k plots are included in their entirety in Appendix A.

Both REPEATED SAMPLING and IDEASEARCH generate exactly n codes, whereas PLANSEARCH generates a variable number of codes, usually ranging on the order of 300 to 400. To compute pass@k, we use the unbiased estimator in Equation 4 [13][1]. If $k > n$, we assume the remaining generations did not pass. To compute pass@k for filtering, we limit the pool of codes to those that are filtered, meaning that both n and c may shrink in size. This can be thought of as a conditional probability, where the condition is that the code passes public tests.

[1]Note that the estimator in Equation 4 theoretically requires that the number of successes follows a binomial distribution. REPEATED SAMPLING and IDEASEARCH obey this, but PLANSEARCH generations may not be independent. See Appendix N for more discussion.

5.3 Results

Our summarized results for REPEATED SAMPLING, IDEASEARCH, and PLANSEARCH can be found in Table 1, Figure 1, and Figure 5. Additionally, we plot our full pass@k curves for all methods, models, and datasets in Appendix A. For sake of easy comparison, we also plot all relative gains compared to REPEATED SAMPLING@1 averaged over all models in Appendix C. For a compute-normalized comparison between REPEATED SAMPLING and PLANSEARCH, see Figure 18.

5.4 Public Test Filtering

Public test filtering is a method which only chooses samples out of the original pool n which pass the public tests. This is particularly useful in settings such as code deployment where executing the full suite of tests may be computationally costly or otherwise undesirable (e.g. in a coding contest where every incorrect submission is penalized). Thus, instead of submitting all n codes, after public test filtering, only codes c_i would be submitted such that $c_i(x_j) = y_j$ for all $j \in \{1, \ldots, u\}$, where $c_i(x)$ refers to the output from running the code on some input x. The primary effect of public test filtering is to shift the pass@k curve leftward, since public test filtering will discard low quality candidate solutions that either fail to compile or fail elementary test cases for the problem.

All problems in MBPP+, HumanEval+, and LiveCodeBench come with a few public tests which are usually used to sanity check any submissions. We can further improve performance by filtering on these public tests before a final submission, as described. Applying public test filtering reduces the number of samples to achieve the same accuracy by tenfold: PLANSEARCH to achieve a 77.1% accuracy on LiveCodeBench after just 20 submissions (pass@20) compared to a pass@200 of 77.0% without using public filtering (see Figure 5). We provide full results for the other datasets in Appendix B.

6. Analysis

Our results suggest that both PLANSEARCH and IDEASEARCH outperform basic sampling by a wide margin (Figures 12, 13, 14), with PLANSEARCH achieving the best score across all methods and models considered. We show the detailed pass@k results for each dataset in Figures 7, 8 and 9. We also compare with Chain-of-Thought [41] in Appendix E. Interestingly, we find that IDEASEARCH performs somewhat better, which we speculate comes from differences in splitting solution sketch into *two* model responses, instead of doing both chain-of-thought and code solution in one model response.

Investigating the differences in specific models, we notice that trends exhibited by the pass@k curves are not uniform across all models; in fact, each curve seems unique. We hypothesize that these differences are in part due to changes in idea diversity, as investigated in Figures 6, 26, 27. From the figures, we can see that our approximate diversity score accounts for much of the variance we see in the relative improvement that arrives from scaling-up inference-time compute. This correlation holds across all methods and models on the same dataset, thus suggesting that diversity score can be used as a proxy to predict for relative pass@k improvement. For further discussion on the specifics of the diversity score, see Section 6.1.

One interesting point of observation is that PLANSEARCH often hurts pass@1 for several models, including most notably Sonnet 3.5 on LiveCodeBench, our best performing combination. Intuitively, this is because increasing the diversity across ideas likely dilutes the probability that any *particular* idea is generated, while simultaneously increasing the chance of having *at least one* correct idea within said pool. Therefore, pass@1 may be slightly lower than usual, yet pass@k will likely surpass "pools" of ideas lacking diversity for this reason. See Figure 48 for a graphical intuition.

Finally, in Table 1 and Figure 1, we present our main results normalized across attempts/completion,

where each search method is allowed k attempts to solve each problem. An alternative method of normalizing across methods is to equalize the amount of compute spent on each method. Since PLANSEARCH and IDEASEARCH first plan out an idea before implementing the final solution, they both spend more compute at inference time per solution generated. In Appendix D, we report the equivalent plots normalized across compute. Our findings are highly similar and suggest that PLANSEARCH outperforms all other methods if sufficient compute is expended at inference time.

6.1 Measuring Diversity

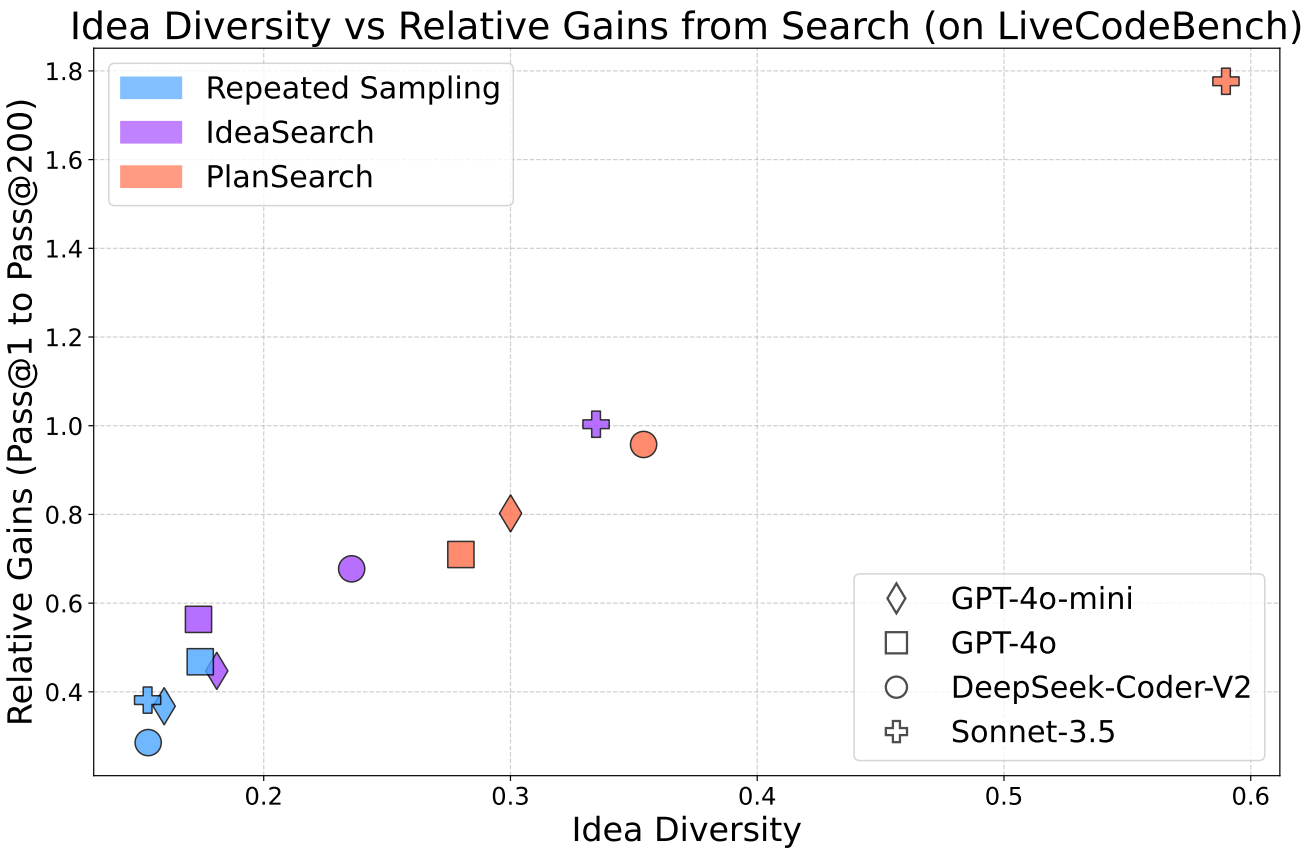


Figure 6: We observe a strong positive correlation between the measured amount of idea diversity in a search algorithm and the resulting improvements due to search (Section 6.1). Diversity score is the probability that GPT-4o-mini believes two randomly selected output codes implement different ideas (higher is more diverse). Our findings suggest that diversity in idea space is essential for effective LLM search.

We find that diversity as measured in idea space is highly predictive of search performance, as measured by the relative improvement between a model/method's pass@1 and its pass@200 (Figure 6). While the most common measure of diversity is entropy [36], entropy is insufficient for a number of reasons for the precise setting of LLMs [21, 45]. As a simple example, consider two different language models, one of which generates minor variations of the same program while another generates a variety of programs with different underlying ideas. Even if both models have the same entropy, the latter model will be significantly better when augmented with search capabilities.

In our setting, we measure diversity by grounding it in *idea* space using a simple pair-matching strategy across all generated programs. Formally, suppose we have a pool of n code generations, $\{c_1, \ldots, c_n\}$. We

assume that each piece of code implements some sketch, which can be thought to exist in some latent 'idea' space. We consider two sketches similar if they are within ϵ of each other in this latent space, for some choice of ϵ. As such, in this space, c_i having a similar idea to c_j and similarly for c_j and c_k *does not imply* c_i and c_k share a similar idea.

To compute the diversity of such a given generation pool, we ask an LLM to judge the similarity of two ideas in the following manner. First, we construct each of the $\binom{n}{2}$ pairs. For each pair (c_i, c_j), we judge (using an LLM) whether both c_i and c_j implement the same idea. We define this as the function $S(c_i, c_j) \in \{0, 1\}$, which evaluates to 1 if c_i and c_j implement the same idea and 0 otherwise. Our overall diversity score for a particular problem is then defined as:

$$D = 1 - \frac{\sum_{i<j} S(c_i, c_j)}{\binom{n}{2}} \tag{1}$$

Models that output programs that all implement the same idea will have a score of $D = 0$, while models that output completely unique programs will have a score of $D = 1$. Overall, a score of D implies that if two codes are chosen at random, the probability that they are the same idea (as measured by the LLM) is D. In Appendix M, we describe this measure in additional mathematical depth.

For a particular method, our reported diversity score is simply the diversity score over all problems in the considered dataset. For computational feasibility, for large n, we instead sample a subset of 40 codes and test all pairs from that subset instead. In order to test code samples, we first backtranslate using an LLM to express the code in natural language before comparing each pair using both the code and the backtranslated idea. We detail the prompts used in Appendix J.5 and use OpenAI's GPT-4o-mini as the supporting LLM.

7. Limitations and Future Work

While PLANSEARCH substantially improves diversity over idea space at inference-time, fundamentally, improvements in diversity should come at the post-training stage. This likely requires re-imagining the post-training pipeline for LLMs around search, instead of the current paradigm optimized for a single correct response. This may require both collecting high quality post-training data that is also sufficiently diverse, and new learning objectives that do not aim solely to maximize the expected reward of a given response. We are optimistic around future work to design significantly improved post-training objectives that maximize both quality and diversity and which specifically optimized to use inference-time compute to maximum effectiveness.

In terms of methodological improvements to PLANSEARCH, PLANSEARCH currently searches all leaf nodes in the search tree uniformly. Because of this, it becomes quickly intractable to go further than a couple levels deep, and in our experiments, we are only able to go two levels down the tree. Several approaches based on Monte-Carlo Tree Search (MCTS), such as Tree of Thought [43] or Reasoning as Planning [19], have suggested that some form of dynamic pruning and expansion of nodes can be very helpful. We are optimistic that PLANSEARCH can be further improved by such methods. Furthermore, PLANSEARCH is a fairly elementary method taking advantage of the paradigm that searching over a *conceptual or idea space* is an effective method to improve diversity, and thus, downstream task performance. It is completely feasible to search at an even higher level of abstraction than observations, which may be used to inject even more diversity into the final generated outputs.

PLANSEARCH and IDEASEARCH tradeoff a slight deterioration of pass@1 performance for a large improvement in pass@k performance. However, in many such cases outside of code generation, it is infeasible to run an LLM-based model for more than a few attempts at most. For example, in Figure 9, PLANSEARCH does not significantly outperform REPEATED SAMPLING until $k \geq 4$.

Fortunately, many filtering algorithms exist, which implicitly bring pass@k (for high k) to pass@1 (or

lower k), i.e. shifting the original pass@k curve leftward. A simple example of this is public test filtering. As seen in Figure 5, pass@1 of filtered PLANSEARCH significantly improves upon pass@1 of base REPEATED SAMPLING, which gets even better as k increases. Moreover, *most to almost all* base models with public test filtering outperform their instruct model variants at pass@1, no matter the dataset (see Appendix I), where clearly base models are known to be worse, yet trading off for somewhat higher diversity. Thus, we argue that there exists a potential for a new paradigm—developing search algorithms which tradeoff pass@1 performance for much stronger pass@k performance, then filtering the promising generated solutions to extract the pass@k *back into* the pass@1.

Additionally, we focus on code generation in this paper and do not consider the applicability of PLANSEARCH to a broader set of domains. One point of importance is that the pass@k metric heavily used throughout code generation may not be as applicable to other domains and a larger focus on selecting the correct solution out of all possible generated candidates may be required, instead of merely generating the correct solution. However, with good filtering methods, which we demonstrate can be simple in nature, pass@k, for medium k, can be effectively brought down to pass@1, emphasizing a similar paradigm of increasing diversity, then strengthening existing filtering methods.

Finally, a natural extension of this work is training the underlying model itself on successful plans and code solutions obtained from PLANSEARCH. This has the potential to distill the pass@k into the pass@1—without inference-time methods like filtering—by reducing the likelihood of the model going down branches of the search tree which do not lead to correct solutions. We believe that such training is likely to significantly improve the model and look forward to future work in this direction.

8. Acknowledgements

We would like to thank Jason Wei, Miles Turpin, Sail Wang, Horace He, Kenneth Li, Celia Chen, Rahul Chalamala, Alan Wu, and Kevin Chang for their helpful comments, suggestions and discussion over the course of this project.

References

[1] Zhiqiang Shen Aidar Myrzakhan, Sondos Mahmoud Bsharat. Open-llm-leaderboard: From multi-choice to open-style questions for llms evaluation, benchmark, and arena. *arXiv preprint arXiv:2406.07545*, 2024.

[2] Anthropic. Claude 3.5 sonnet. `https://www.anthropic.com/news/claude-3-5-sonnet`, June 2024.

[3] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021. URL `https://arxiv.org/abs/2108.07732`.

[4] Anton Bakhtin, David J. Wu, Adam Lerer, Jonathan Gray, Athul Paul Jacob, Gabriele Farina, Alexander H. Miller, and Noam Brown. Mastering the Game of No-Press Diplomacy via Human-Regularized Reinforcement Learning and Planning, October 2022. URL `http://arxiv.org/abs/2210.05492`. arXiv:2210.05492 [cs].

[5] Hritik Bansal, Arian Hosseini, Rishabh Agarwal, Vinh Q. Tran, and Mehran Kazemi. Smaller, weaker, yet better: Training llm reasoners via compute-optimal sampling, 2024. URL `https://arxiv.org/abs/2408.16737`.

[6] Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V. Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling, 2024. URL `https://arxiv.org/abs/2407.21787`.

[7] Noam Brown and Tuomas Sandholm. Superhuman AI for heads-up no-limit poker: Libratus beats top professionals. *Science*, 359(6374):418–424, 2018. Publisher: American Association for the Advancement of Science.

[8] Noam Brown and Tuomas Sandholm. Superhuman AI for multiplayer poker. *Science*, 365(6456): 885–890, 2019. Publisher: American Association for the Advancement of Science.

[9] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 1877–1901. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

[10] Murray Campbell, A.Joseph Hoane, and Feng hsiung Hsu. Deep blue. *Artificial Intelligence*, 134(1):57–83, 2002. ISSN 0004-3702. doi: https://doi.org/10.1016/S0004-3702(01)00129-1. URL https://www.sciencedirect.com/science/article/pii/S0004370201001291.

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

[12] Lingjiao Chen, Jared Quincy Davis, Boris Hanin, Peter Bailis, Ion Stoica, Matei Zaharia, and James Zou. Are more llm calls all you need? towards scaling laws of compound inference systems, 2024. URL https://arxiv.org/abs/2403.02419.

[13] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating Large Language Models Trained on Code, July 2021. URL http://arxiv.org/abs/2107.03374. arXiv:2107.03374 [cs].

[14] Wei-Lin Chiang, Lianmin Zheng, Ying Sheng, Anastasios Nikolas Angelopoulos, Tianle Li, Dacheng Li, Hao Zhang, Banghua Zhu, Michael Jordan, Joseph E. Gonzalez, and Ion Stoica. Chatbot arena: An open platform for evaluating llms by human preference, 2024. URL https://arxiv.org/abs/2403.04132.

[15] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[16] Sergey Edunov, Myle Ott, Michael Auli, and David Grangier. Understanding back-translation at scale, 2018. URL https://arxiv.org/abs/1808.09381.

[17] FAIR, Anton Bakhtin, Noam Brown, Emily Dinan, Gabriele Farina, Colin Flaherty, Daniel Fried, Andrew Goff, Jonathan Gray, Hengyuan Hu, Athul Paul Jacob, Mojtaba Komeili, Karthik Konath,

Minae Kwon, Adam Lerer, Mike Lewis, Alexander H. Miller, Sasha Mitts, Adithya Renduchintala, Stephen Roller, Dirk Rowe, Weiyan Shi, Joe Spisak, Alexander Wei, David Wu, Hugh Zhang, and Markus Zijlstra. Human-level play in the game of Diplomacy by combining language models with strategic reasoning. *Science*, 378(6624):1067–1074, December 2022. doi: 10.1126/science.ade9097. URL https://www.science.org/doi/10.1126/science.ade9097. Publisher: American Association for the Advancement of Science.

[18] Markus Freitag and Yaser Al-Onaizan. Beam search strategies for neural machine translation. In *Proceedings of the First Workshop on Neural Machine Translation*. Association for Computational Linguistics, 2017. doi: 10.18653/v1/w17-3207. URL http://dx.doi.org/10.18653/v1/W17-3207.

[19] Shibo Hao, Yi Gu, Haodi Ma, Joshua Jiahua Hong, Zhen Wang, Daisy Zhe Wang, and Zhiting Hu. Reasoning with Language Model is Planning with World Model, May 2023. URL http://arxiv.org/abs/2305.14992. arXiv:2305.14992 [cs].

[20] Shibo Hao, Yi Gu, Haodi Ma, Joshua Jiahua Hong, Zhen Wang, Daisy Zhe Wang, and Zhiting Hu. Reasoning with language model is planning with world model, 2023. URL https://arxiv.org/abs/2305.14992.

[21] Tatsunori B. Hashimoto, Hugh Zhang, and Percy Liang. Unifying Human and Statistical Evaluation for Natural Language Generation. *North American Association for Computational Linguistics (NAACL).*, April 2019. URL http://arxiv.org/abs/1904.02792. arXiv: 1904.02792.

[22] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

[23] Andy L. Jones. Scaling scaling laws with board games, 2021. URL https://arxiv.org/abs/2104.03113.

[24] Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy Liang. Spoc: Search-based pseudocode to code, 2019. URL https://arxiv.org/abs/1906.04908.

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

[26] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=1qvx610Cu7.

[27] Drew M. McDermott. The 1998 ai planning systems competition. *AI Magazine*, 21(2):35, Jun. 2000. doi: 10.1609/aimag.v21i2.1506. URL https://ojs.aaai.org/aimagazine/index.php/aimagazine/article/view/1506.

[28] Aidan McLaughlin. AI Search: The Bitter-er Lesson, 2024. Accessed on September 3, 2024.

[29] Sidharth Mudgal, Jong Lee, Harish Ganapathy, YaGuang Li, Tao Wang, Yanping Huang, Zhifeng Chen, Heng-Tze Cheng, Michael Collins, Trevor Strohman, Jilin Chen, Alex Beutel, and Ahmad Beirami. Controlled decoding from language models, 2024. URL https://arxiv.org/abs/2310.17022.

[30] OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Jan Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr H. Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine B. Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, C. J. Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. GPT-4 Technical Report, March 2024. URL `http://arxiv.org/abs/2303.08774`. arXiv:2303.08774 [cs].

[31] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

[32] Hieu Pham, Xinyi Wang, Yiming Yang, and Graham Neubig. Meta back-translation, 2021. URL https://arxiv.org/abs/2102.07847.

[33] Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems*, 36, 2024.

[34] Stuart Russell and Peter Norvig. Artificial intelligence: a modern approach. 2002.

[35] Rico Sennrich, Barry Haddow, and Alexandra Birch. Improving neural machine translation models with monolingual data, 2016. URL https://arxiv.org/abs/1511.06709.

[36] Claude E Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27(3): 379–423, 623–656, 1948.

[37] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 529(7587):484–489, January 2016. ISSN 0028-0836, 1476-4687. doi: 10.1038/nature16961. URL http://www.nature.com/articles/nature16961.

[38] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, and others. Mastering the Game of Go Without Human Knowledge. *Nature*, 550(7676):354–359, 2017. Publisher: Nature Publishing Group.

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

[40] Richard S Sutton. The bitter lesson. *Incomplete Ideas*, 2019. URL http://www.incompleteideas.net/IncIdeas/BitterLesson.html.

[41] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. arXiv, 2022. doi: 10.48550/arXiv.2201.11903. URL http://arxiv.org/abs/2201.11903. arXiv:2201.11903 [cs].

[42] Yangzhen Wu, Zhiqing Sun, Shanda Li, Sean Welleck, and Yiming Yang. An empirical analysis of compute-optimal inference for problem-solving with language models, 2024. URL https://arxiv.org/abs/2408.00724.

[43] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of Thoughts: Deliberate Problem Solving with Large Language Models, May 2023. URL http://arxiv.org/abs/2305.10601. arXiv:2305.10601 [cs].

[44] Dan Zhang, Sining Zhoubian, Ziniu Hu, Yisong Yue, Yuxiao Dong, and Jie Tang. Rest-mcts*: Llm self-training via process reward guided tree search, 2024. URL https://arxiv.org/abs/2406.03816.

[45] Hugh Zhang, Daniel Duckworth, Daphne Ippolito, and Arvind Neelakantan. Trading off diversity and quality in natural language generation. In *Proceedings of the workshop on human evaluation of NLP systems (HumEval)*, pp. 25–33, Online, April 2021. Association for Computational Linguistics. URL https://aclanthology.org/2021.humeval-1.3.

[46] Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B. Tenenbaum, and Chuang Gan. Planning with large language models for code generation. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=Lr8cOOtYbfL.

Appendix

A. Full Pass@K curves for All Models and All Benchmarks

See Figures 7, 8, 9. We plot all models and methods on HumanEval+, MBPP+ [26], and LiveCodeBench [22], respectively.

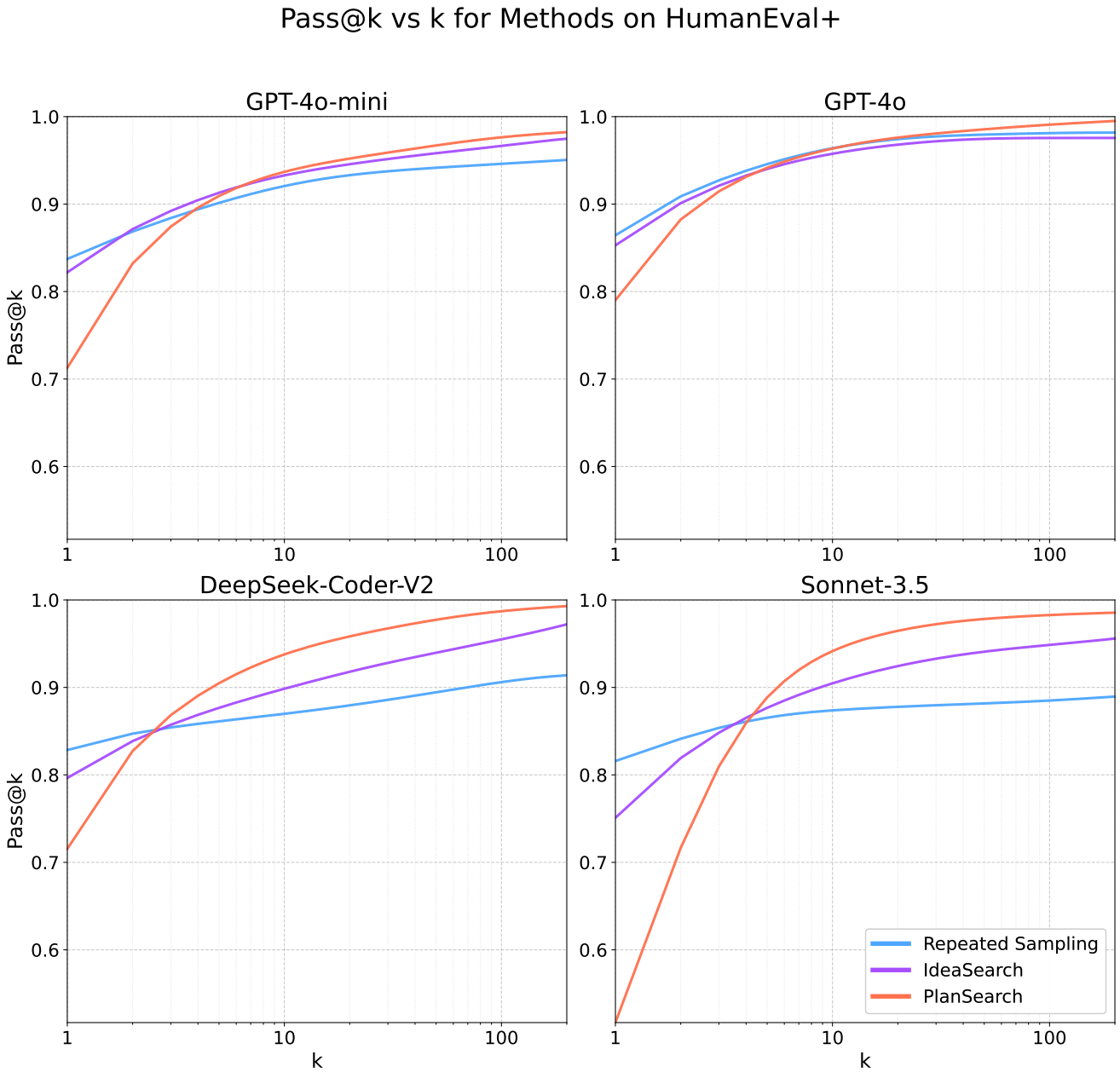


Figure 7: Pass@k performance of all models and methods on HumanEval+, plotted over $k \in \{1, \ldots, 200\}$.

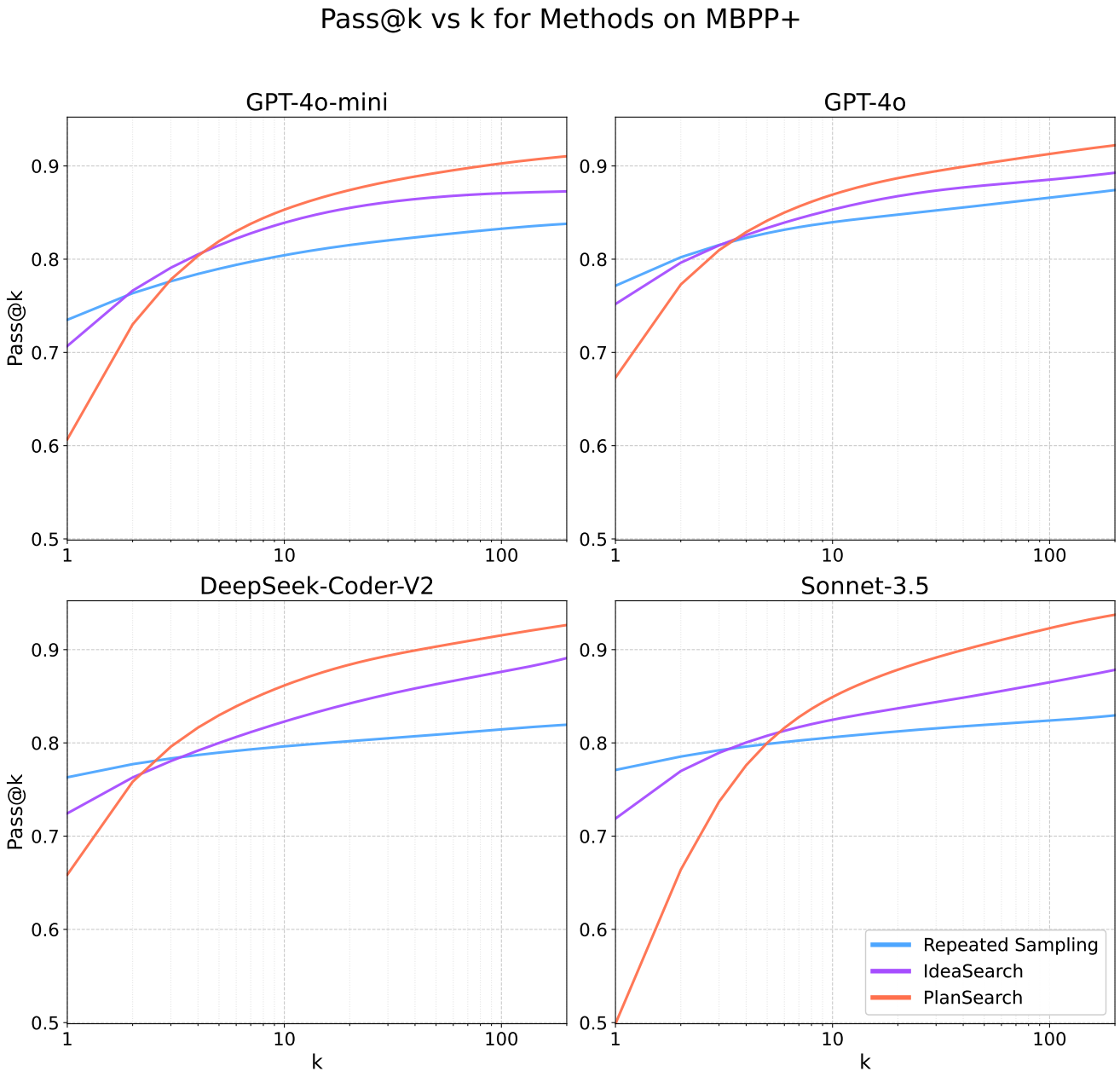


Figure 8: Pass@k performance of all models and methods on MBPP+, plotted over $k \in \{1, \ldots, 200\}$.

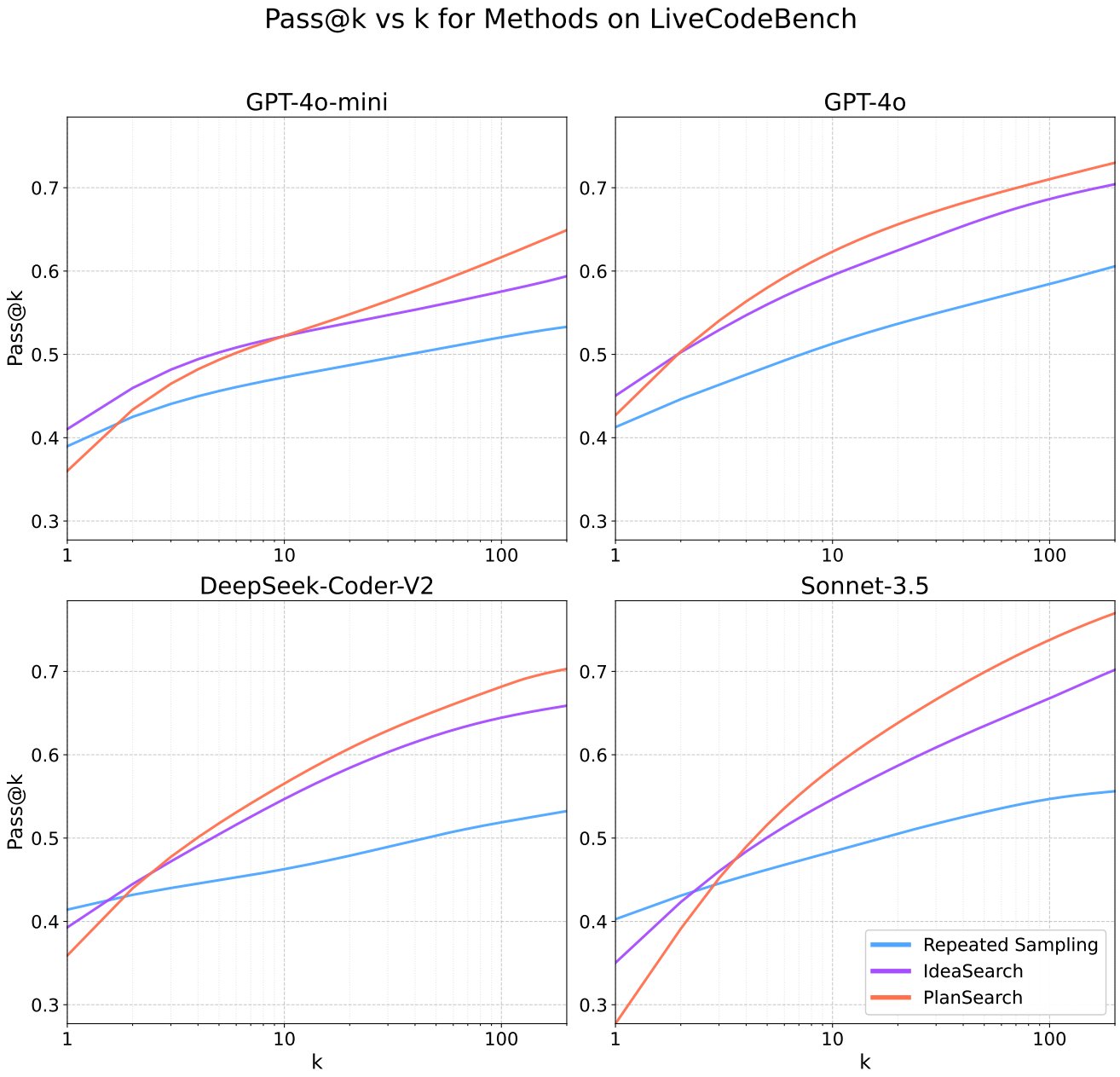


Figure 9: Pass@k performance of all models and methods on LiveCodeBench, plotted over $k \in \{1, \ldots, 200\}$.

B. Full Pass@k Curves with Public Filtering

See Figures 10, 11, 5. We plot all models and methods with public test filtering on HumanEval+, MBPP+ [26], and LiveCodeBench [22], respectively.

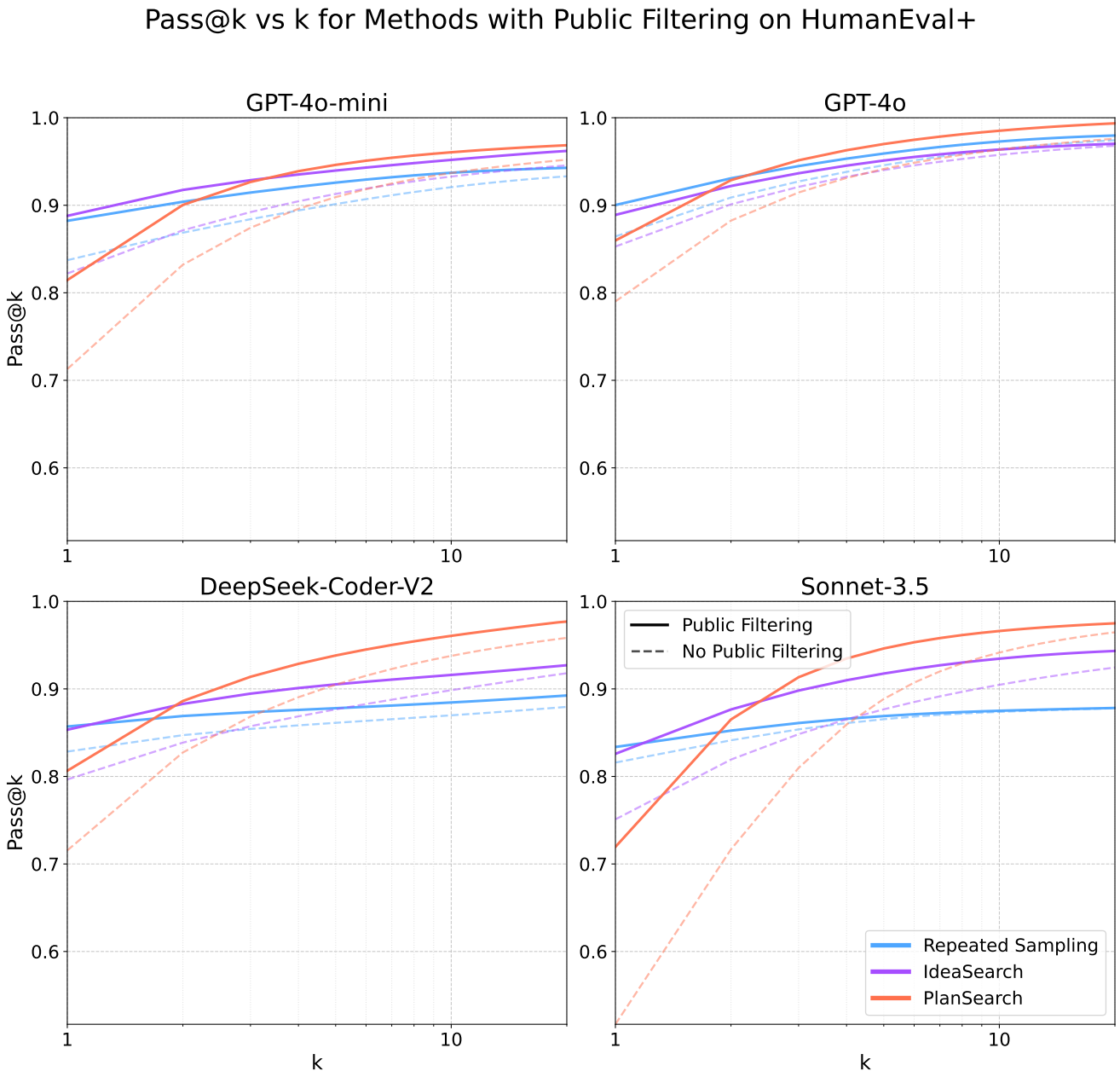


Figure 10: Pass@k performance of all models and methods on HumanEval+, with public test filtering, plotted over $k \in \{1, \ldots, 20\}$. Note that dotted lines are provided for reference of the base method pass@k before filtering.

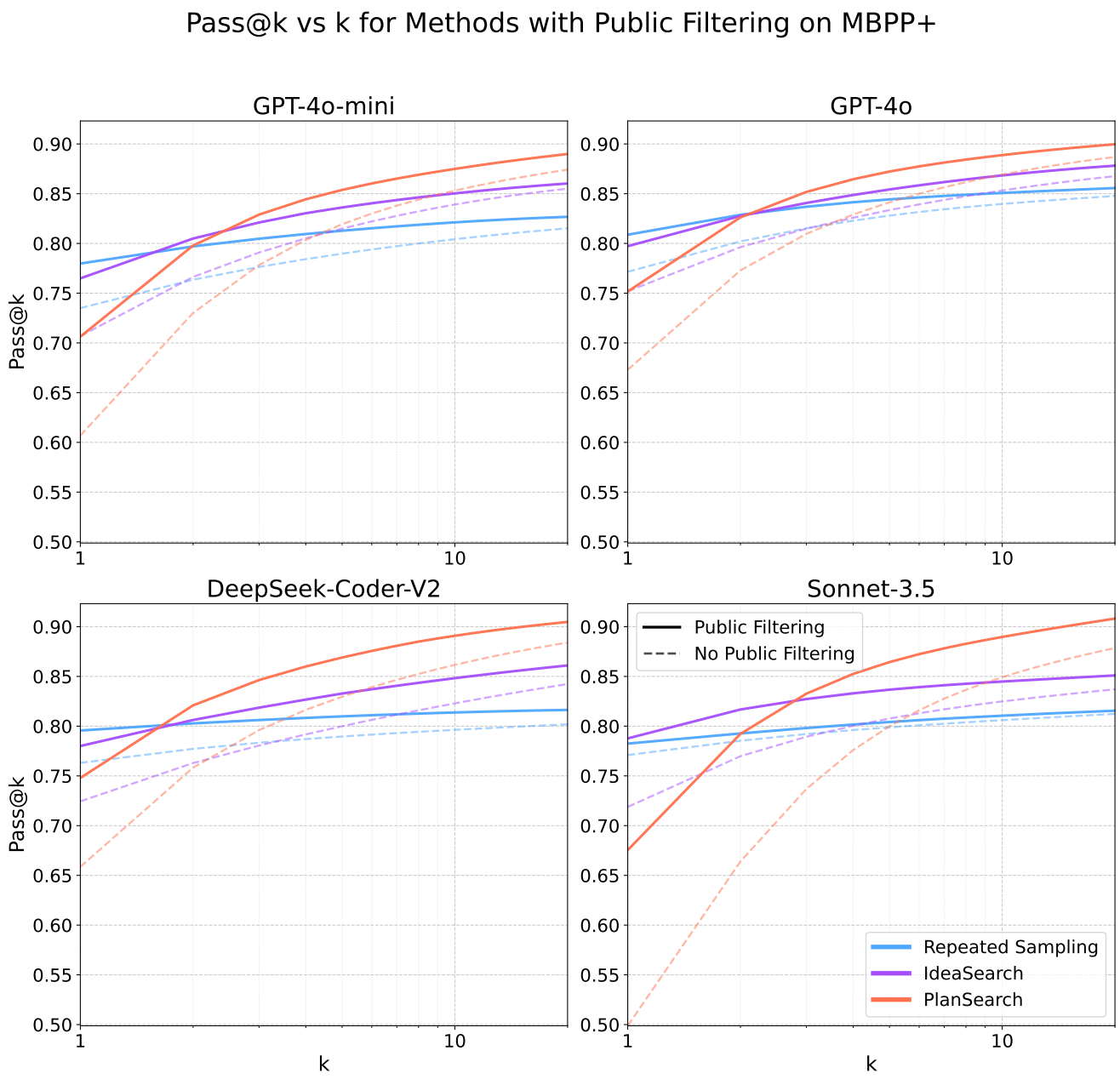


Figure 11: Pass@k performance of all models and methods on MBPP+, with public test filtering, plotted over $k \in \{1, \ldots, 20\}$. Note that dotted lines are provided for reference of the base method pass@k before filtering.

C. Average Relative Improvements

See Figures 12, 13, 14. To create these graphs, the relative improvements of each point on all pass@k curves are computed and compared to the respective pass@1 of REPEATED SAMPLING. Then these values are averaged over all models, so that there is one curve per method per dataset. The datasets are HumanEval+, MBPP+ [26], and LiveCodeBench [22], respectively. For the public test filtered versions, see Figures 15, 16, 17.

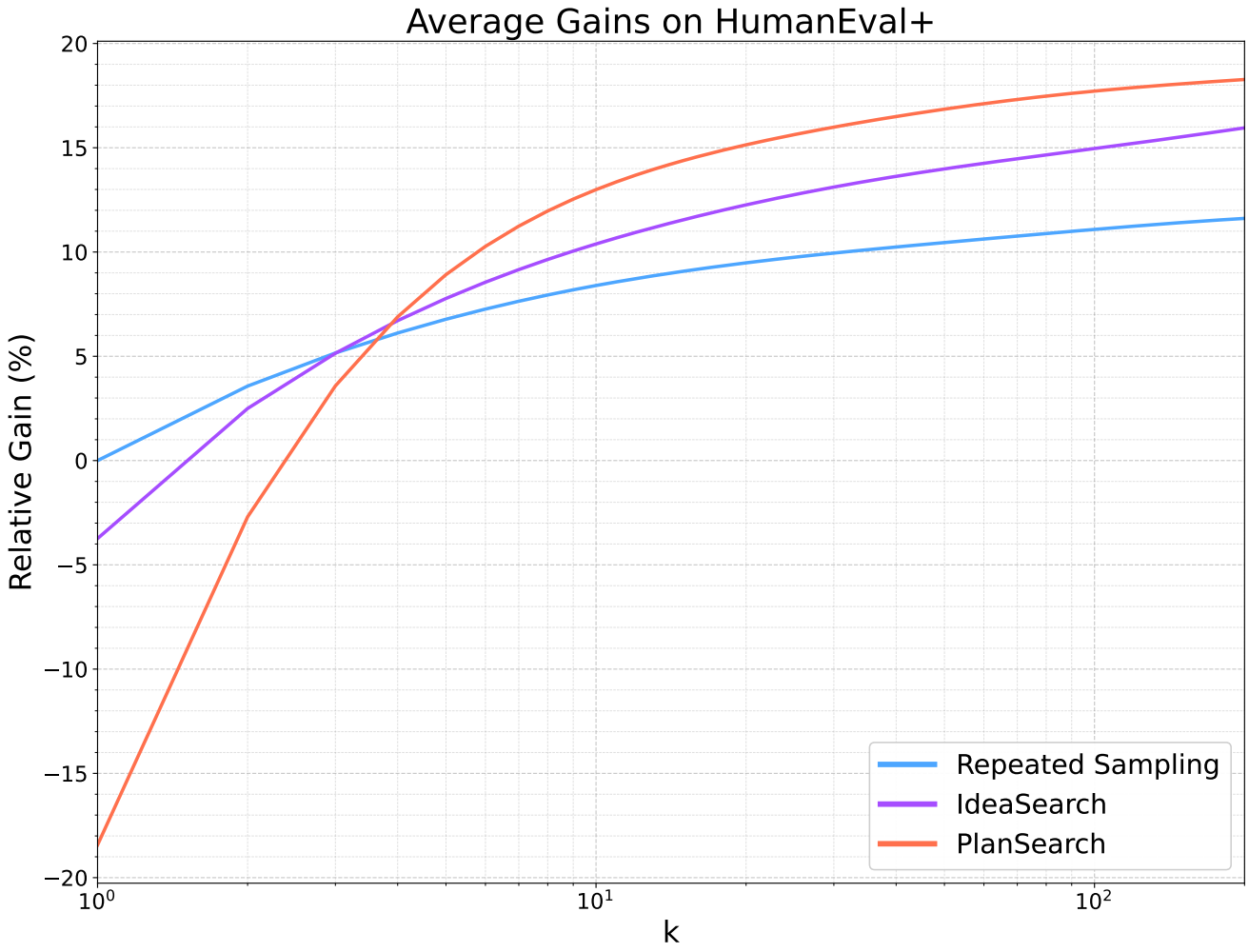


Figure 12: Performance gain over REPEATED SAMPLING@1 averaged over all models on HumanEval+, plotted over $k \in \{1, \ldots, 200\}$.

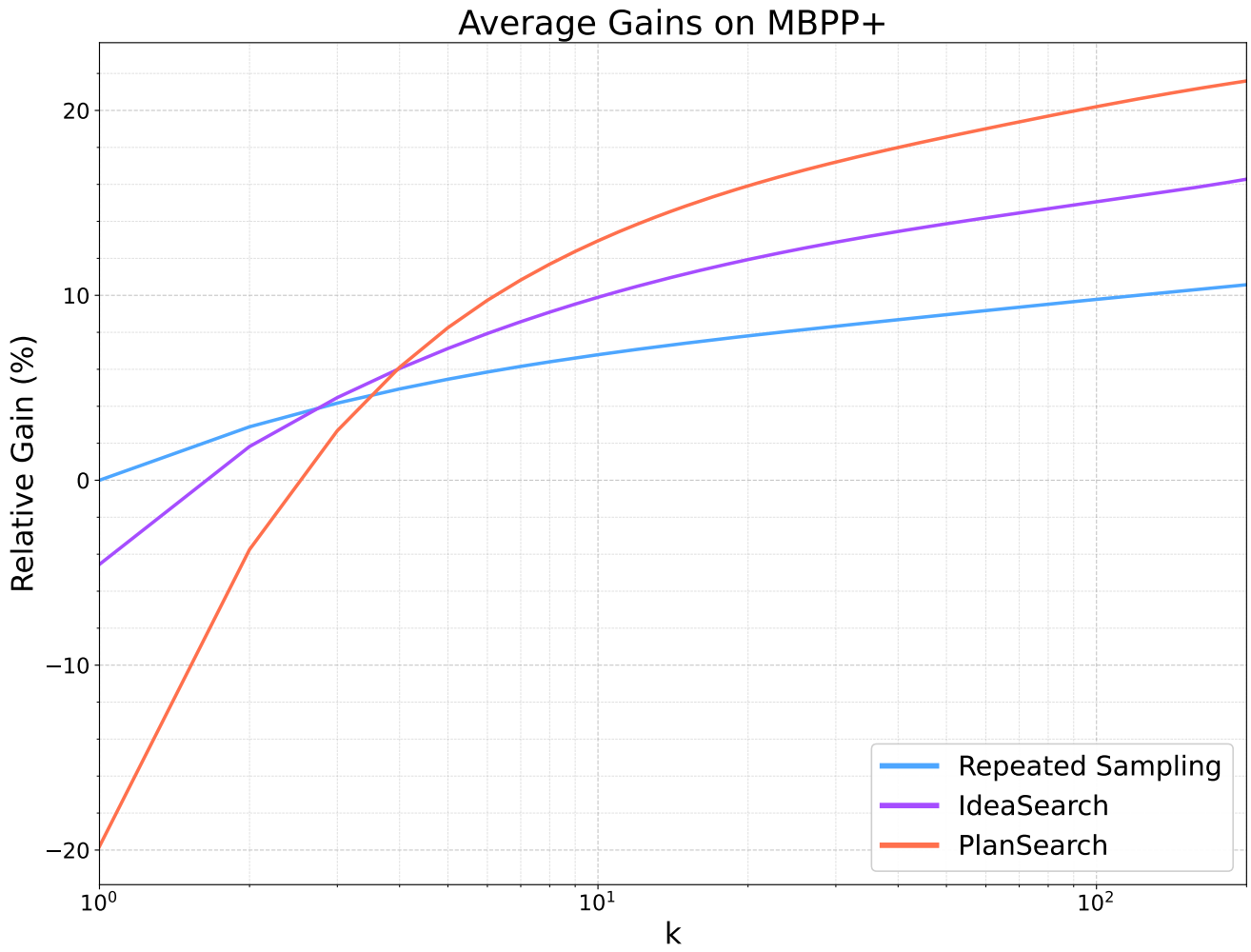


Figure 13: Performance gain over REPEATED SAMPLING@1 averaged over all models on MBPP+, plotted over $k \in \{1, \ldots, 200\}$.

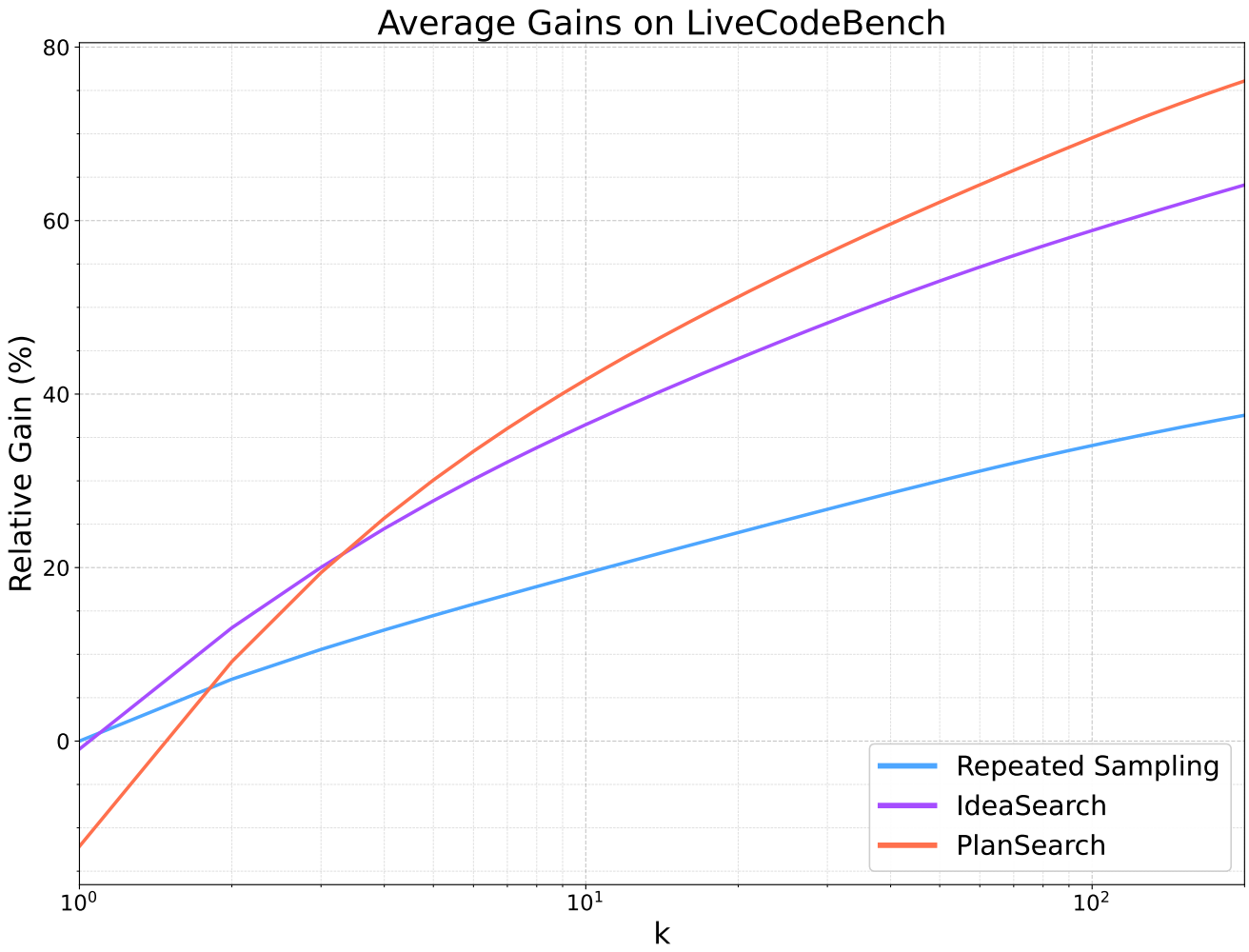


Figure 14: Performance gain over REPEATED SAMPLING@1 averaged over all models on LiveCodeBench, plotted over $k \in \{1, \ldots, 200\}$.

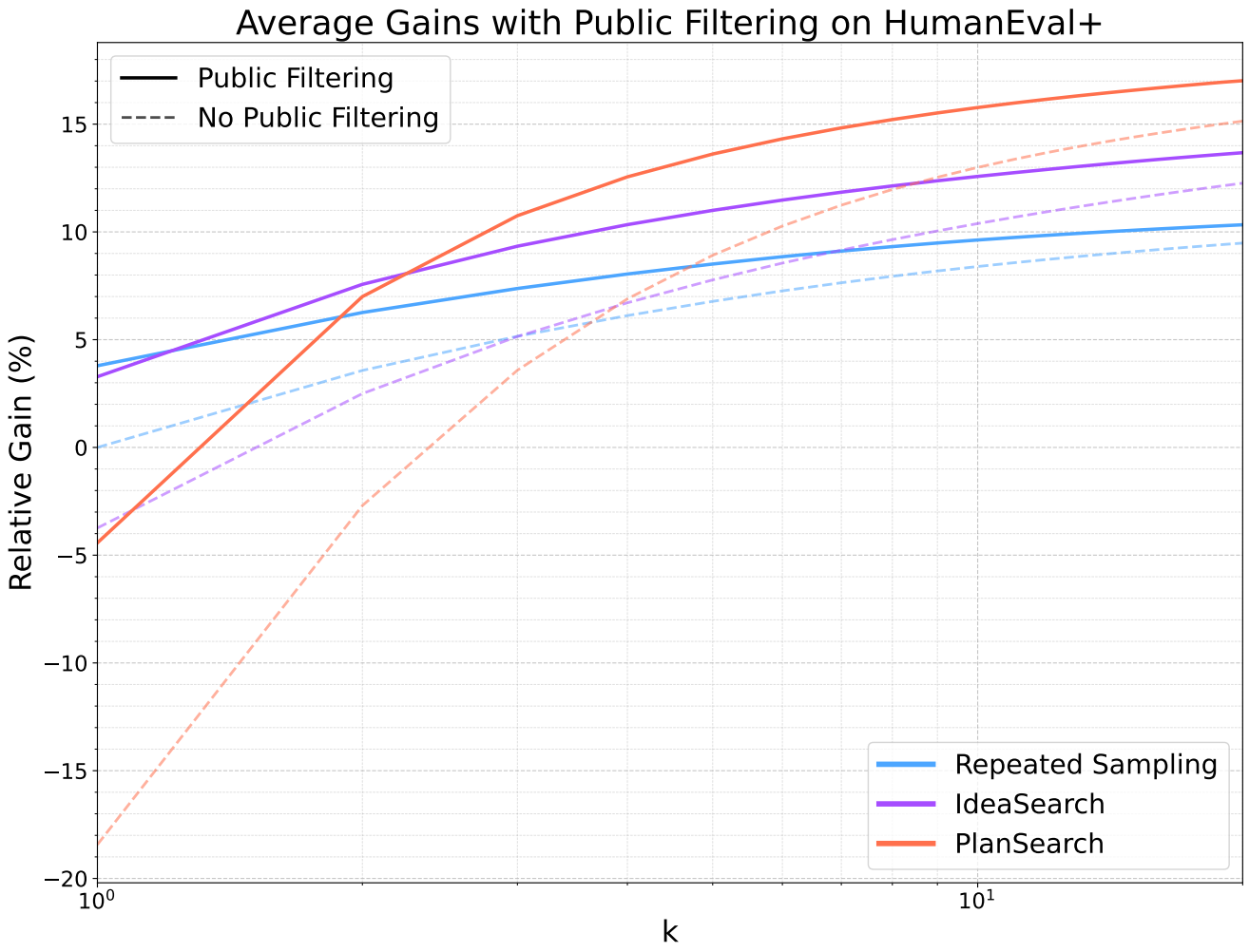


Figure 15: Average performance gain over all models of methods with public test filtering compared to REPEATED SAMPLING@1, plotted over $k \in \{1, \ldots, 20\}$. Note that dotted lines are provided for reference of the base method pass@k (before filtering).

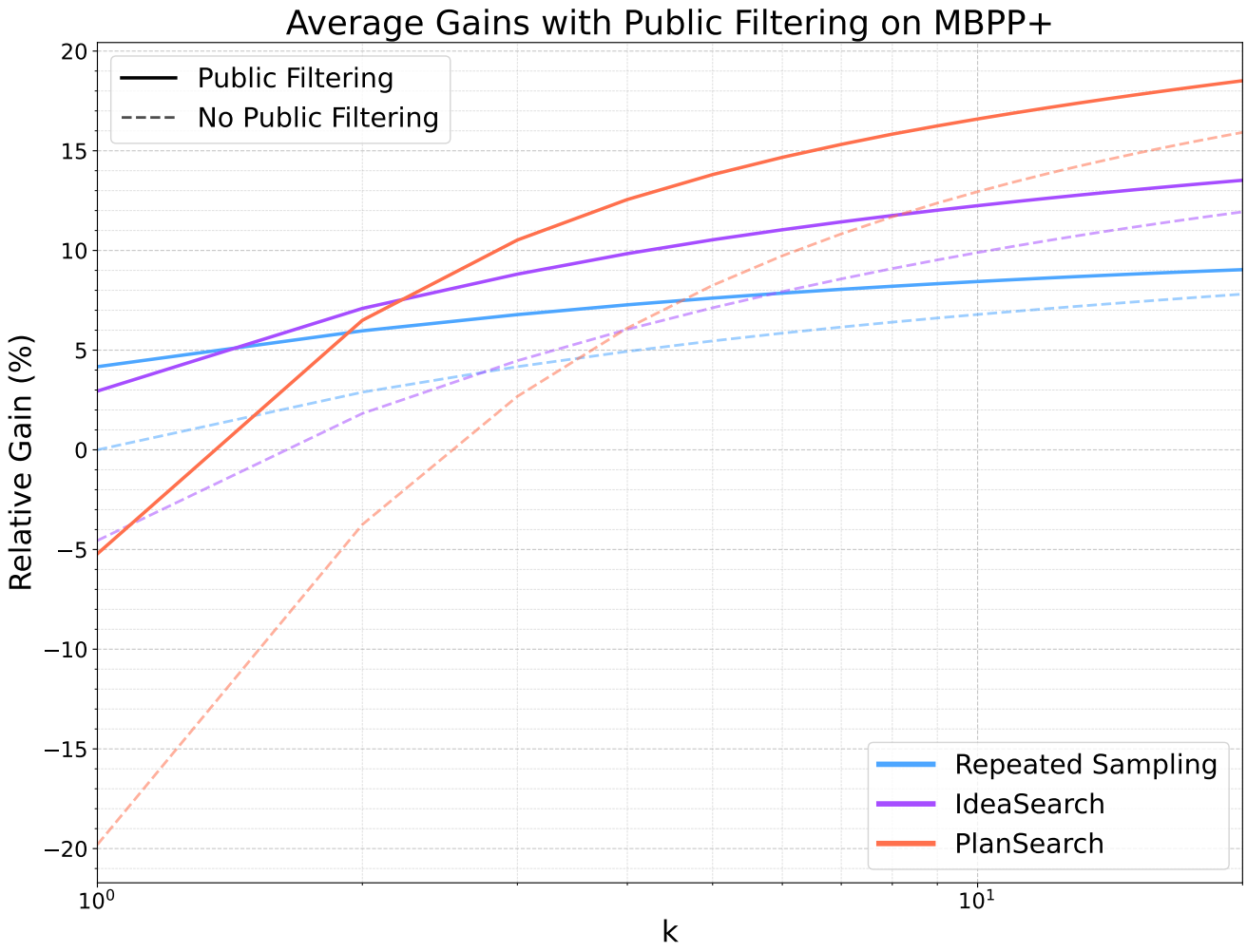


Figure 16: Average performance gain over all models of methods with public test filtering compared to REPEATED SAMPLING@1, plotted over $k \in \{1, \ldots, 20\}$. Note that dotted lines are provided for reference of the base method pass@k (before filtering).

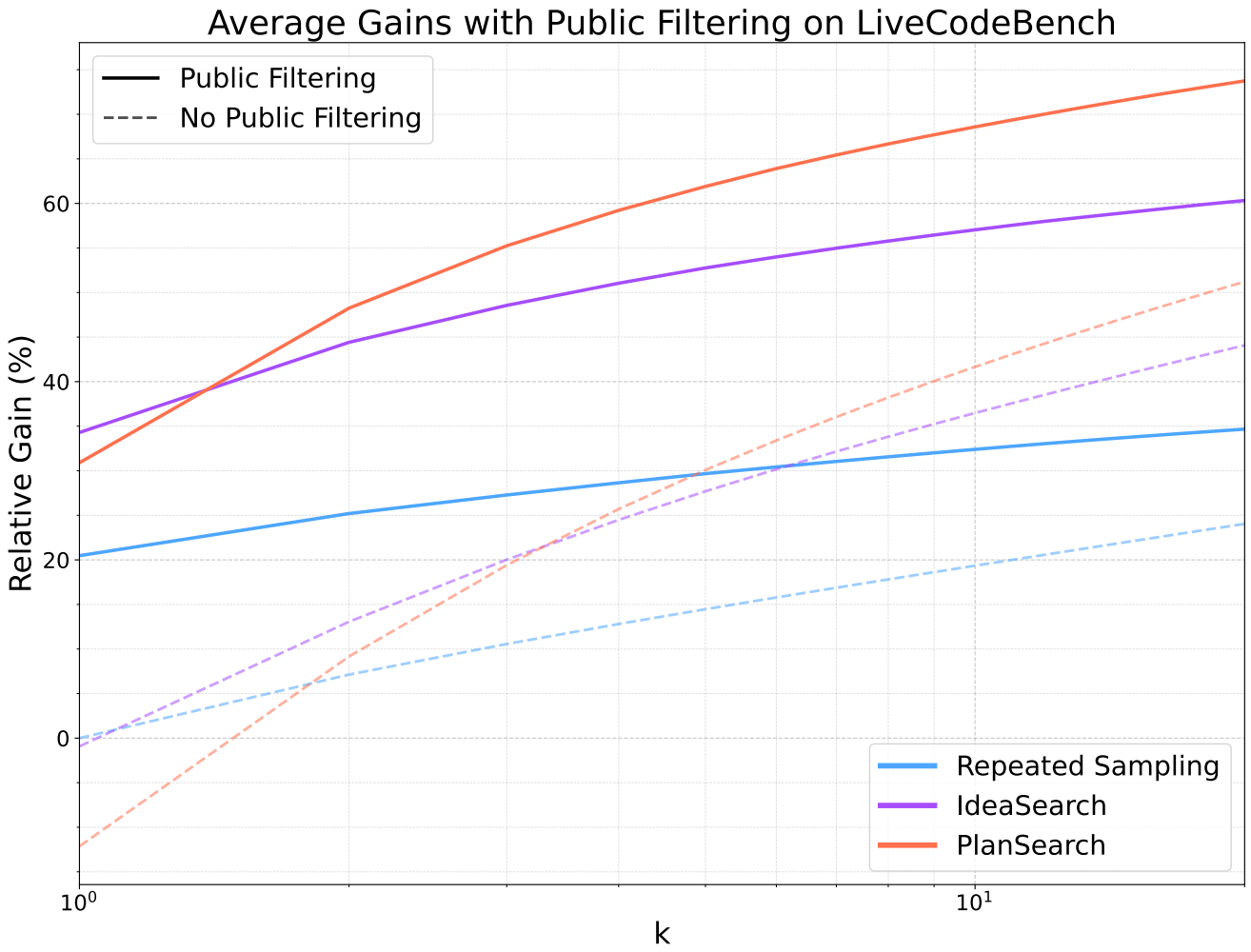


Figure 17: Average performance gain over all models of methods with public test filtering compared to REPEATED SAMPLING@1, plotted over $k \in \{1, \ldots, 20\}$. Note that dotted lines are provided for reference of the base method pass@k (before filtering).

D. Compute Normalized Pass@K Graphs

See Figure 18. For each run of a method in Appendix A, we compute the number of generated tokens needed per completion, per problem, independently on each dataset. Then, we average across all datasets to obtain 244 generated tokens per completion per problem for REPEATED SAMPLING, and 1,428 generated tokens per completion per problem for PLANSEARCH.

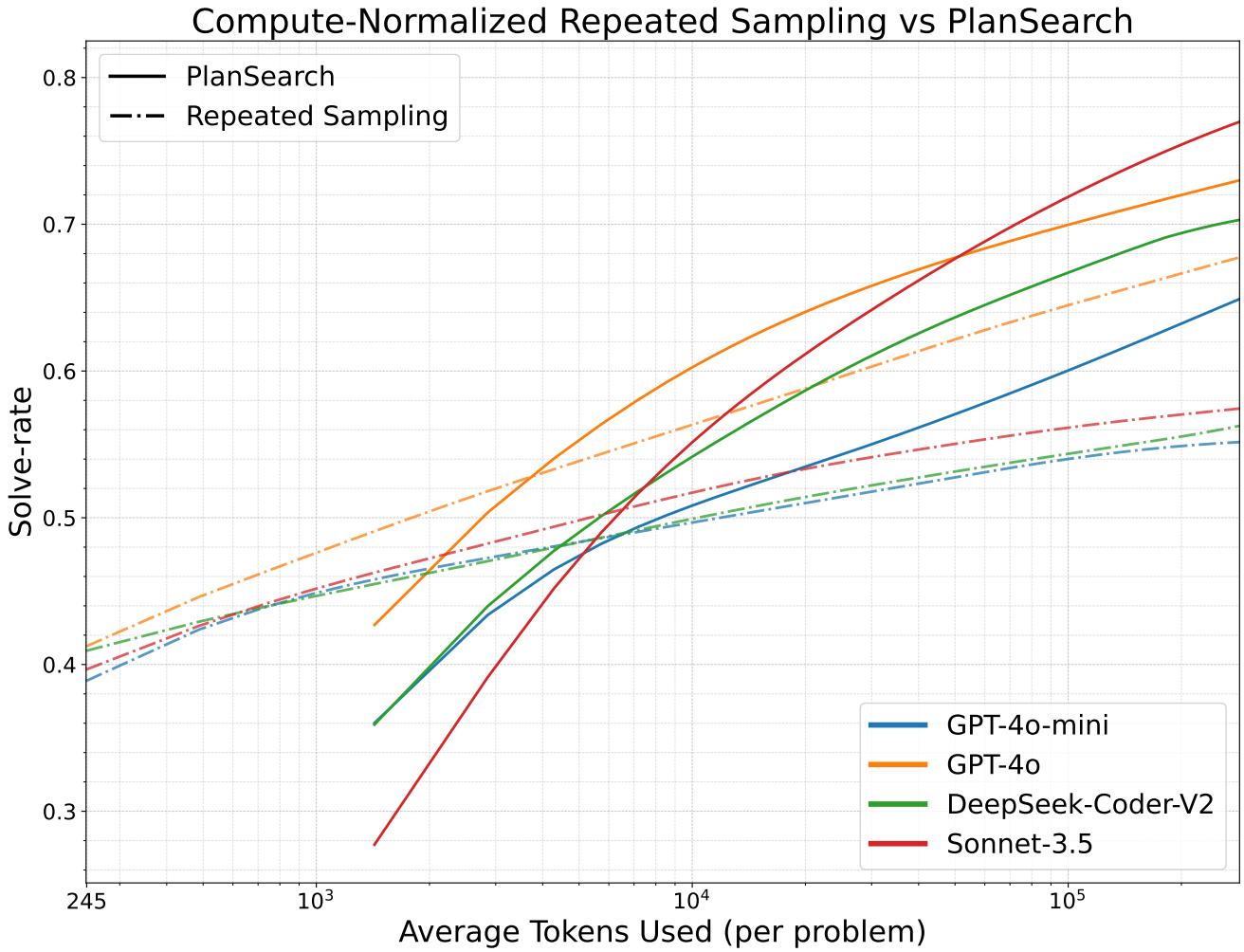


Figure 18: Normalized pass@k by average tokens used per problem. REPEATED SAMPLING uses roughly 244 tokens per completion per problem, and PLANSEARCH uses roughly 1428 tokens per completion per problem. When we normalize compute across methods, we find that PLANSEARCH begins to be more effective than repeated sampling if the user is willing to sample at least 10,000 tokens per problem.

E. Comparison with Chain-of-Thought

See Figures 19, 20, 21, which are run on LiveCodeBench [22], MBPP+, and HumanEval+ [26], respectively. These are the same plots as Appendix A, with CoT [41]. See Figures 22, 23, 24 for the public test filtered versions.

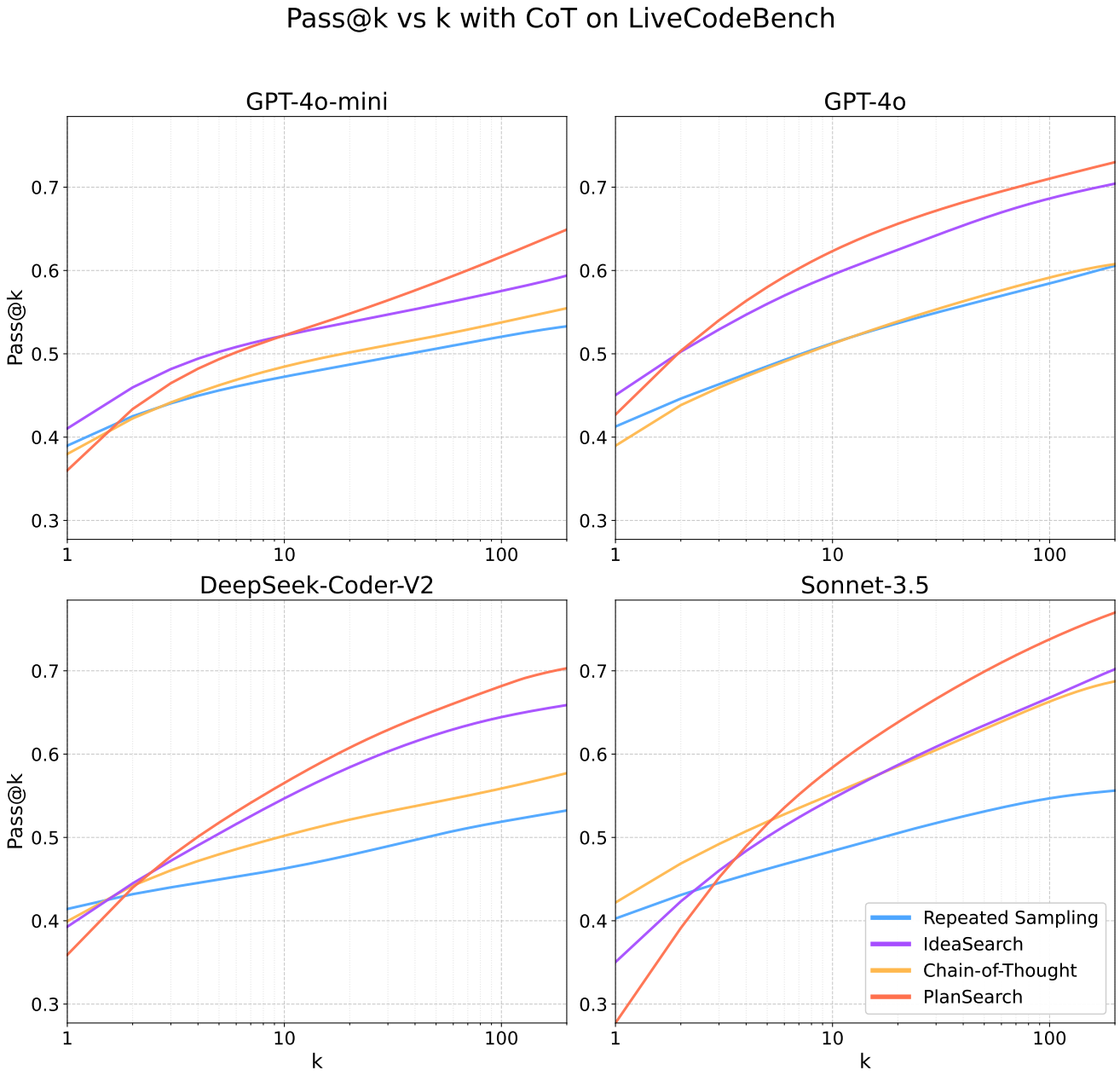


Figure 19: Pass@k graphs on LiveCodeBench, with the Chain-of-Thought baseline.

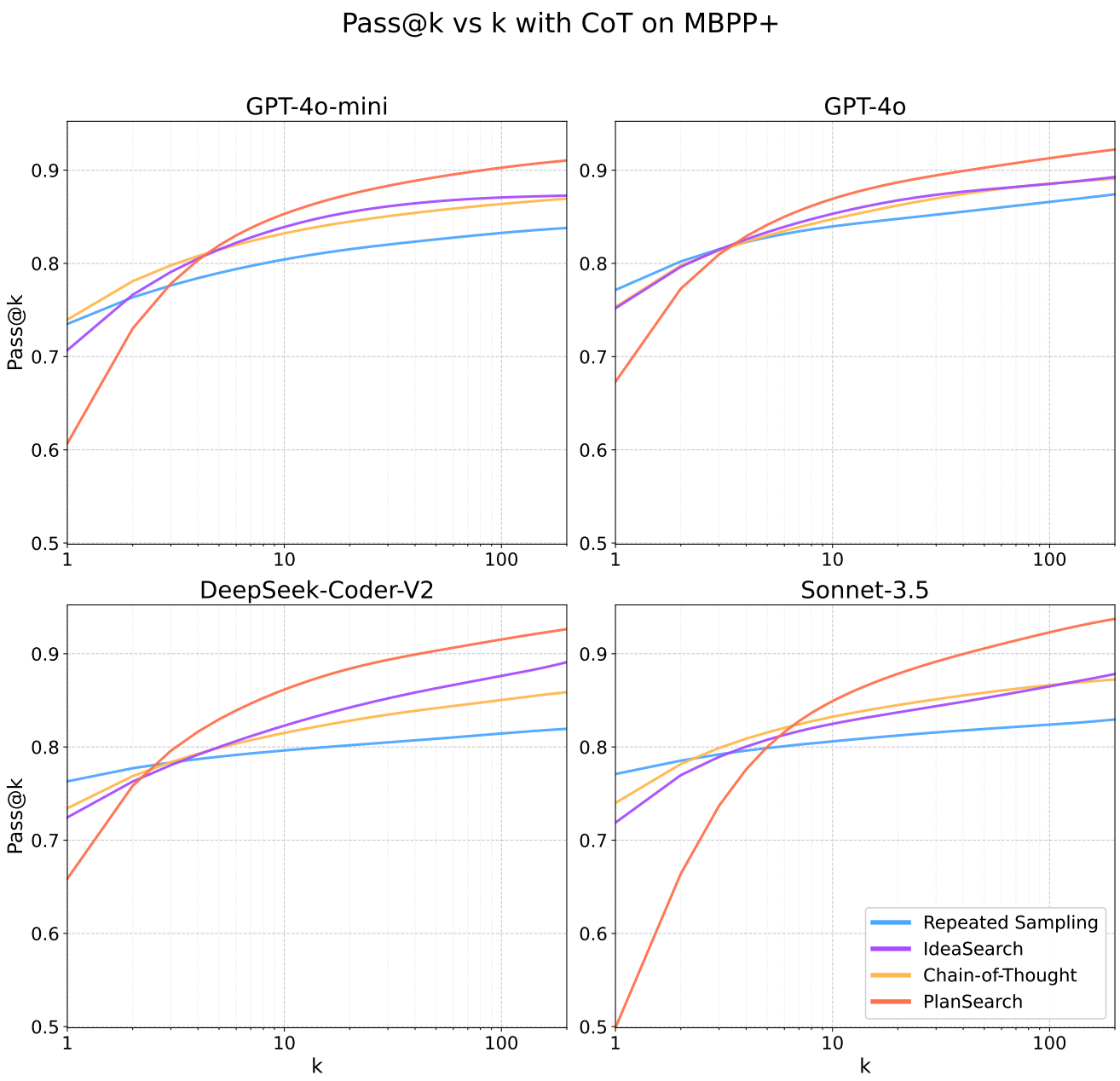


Figure 20: Pass@k graphs on MBPP+, with the Chain-of-Thought baseline.

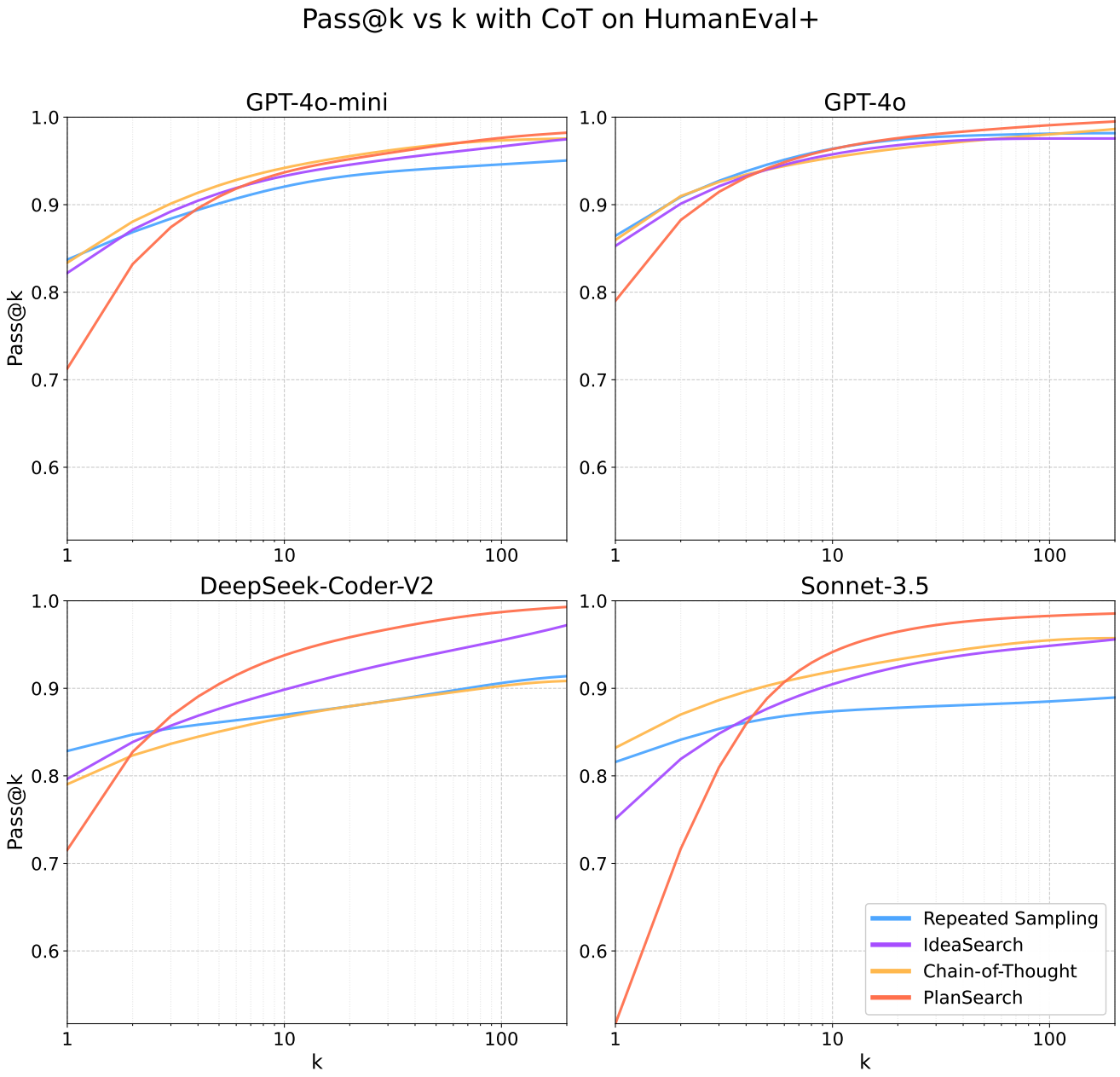


Figure 21: Pass@k graphs on HumanEval+, with the Chain-of-Thought baseline.

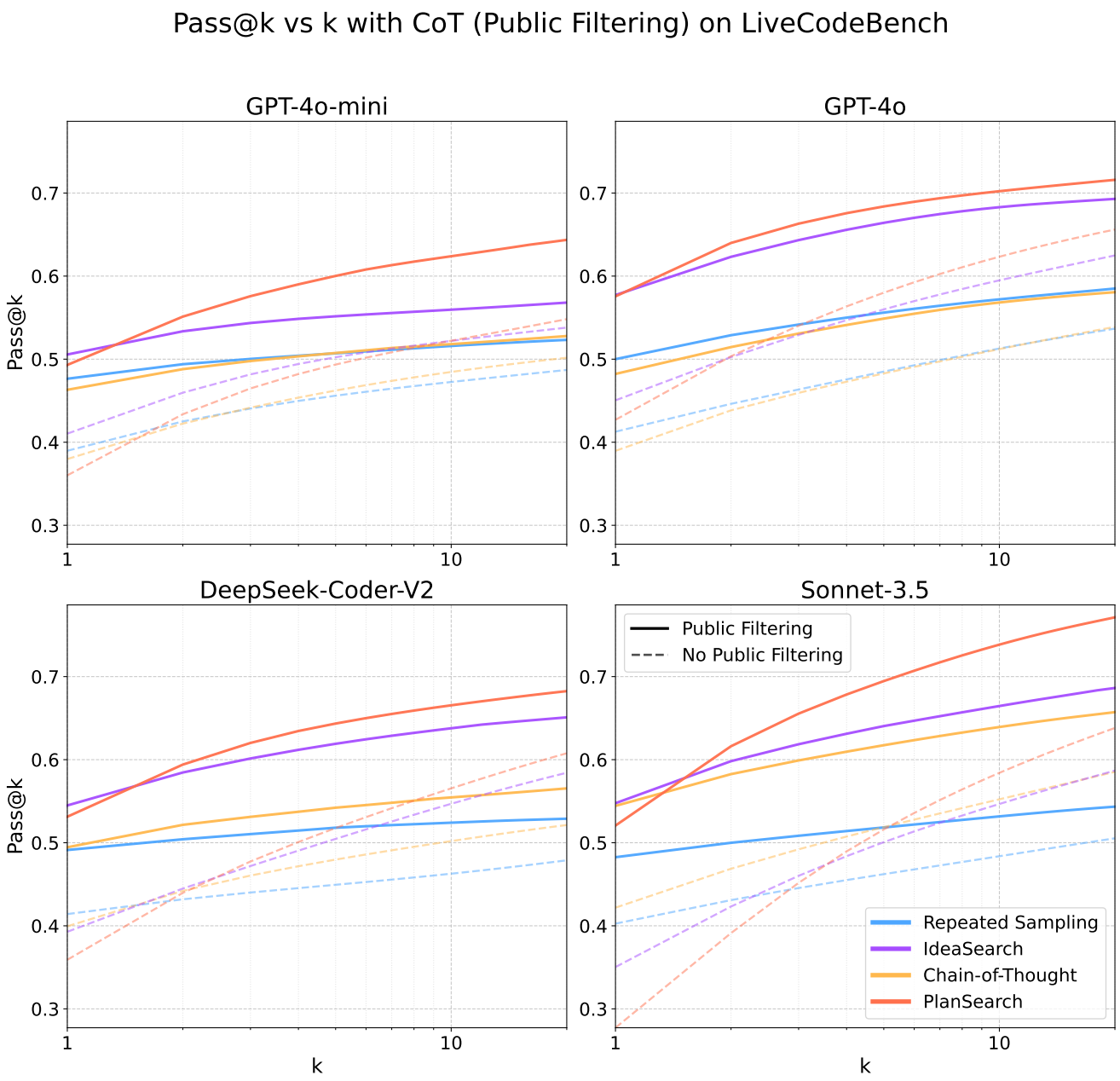


Figure 22: Pass@k graphs on LiveCodeBench, with the Chain-of-Thought baseline and public filtering.

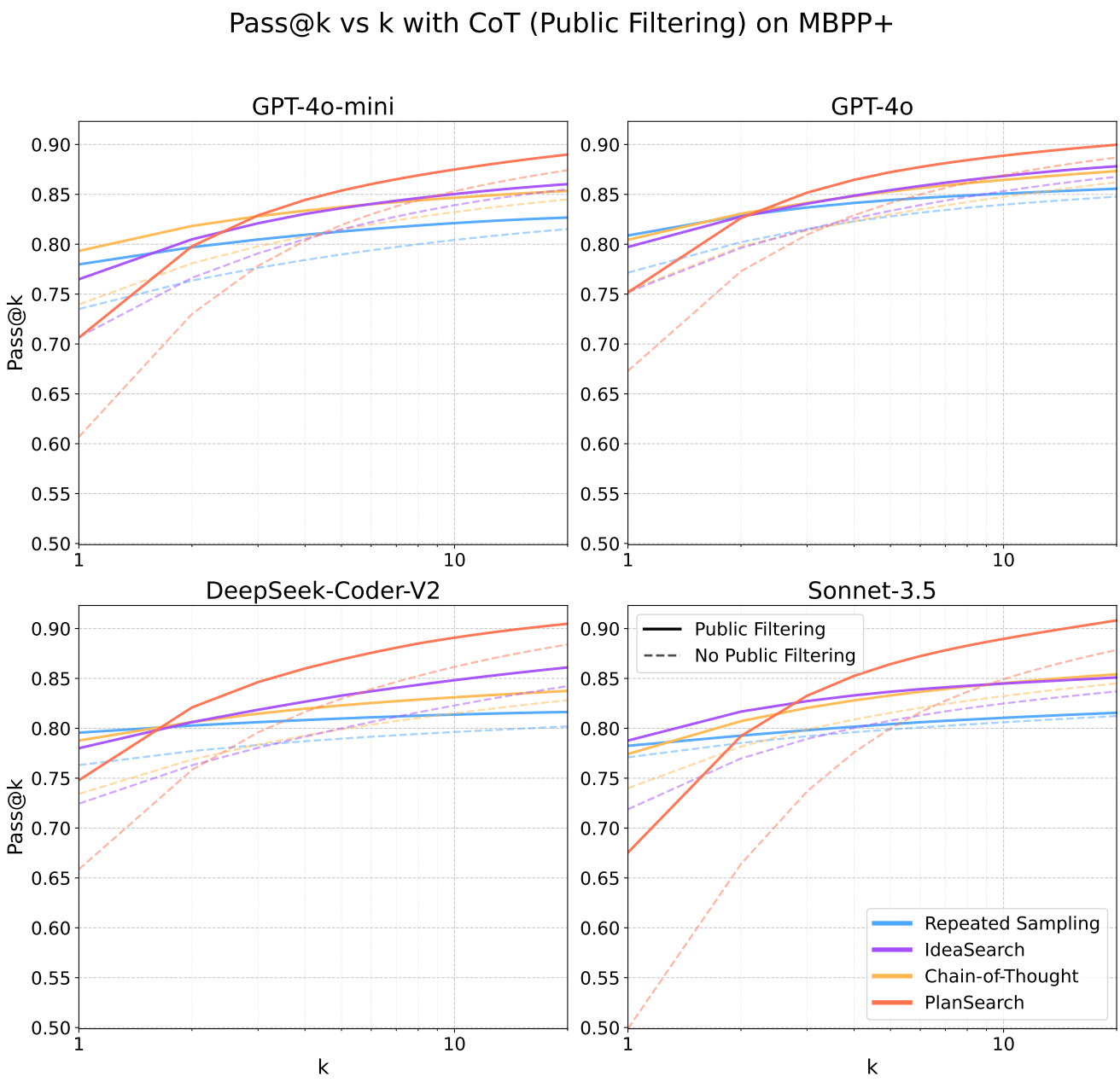


Figure 23: Pass@k graphs on MBPP+, with the Chain-of-Thought baseline and public filtering.

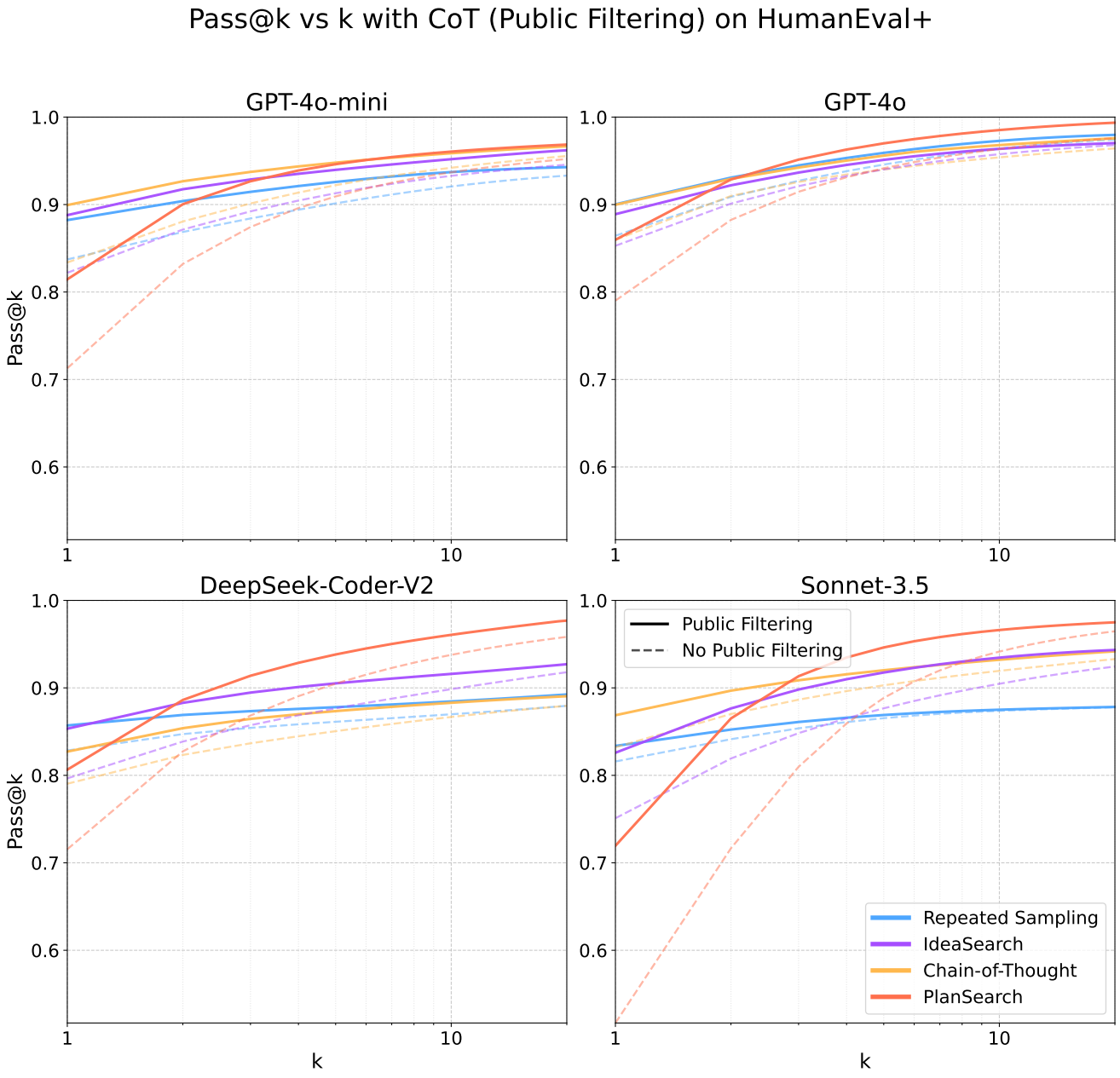


Figure 24: Pass@k graphs on HumanEval+, with the Chain-of-Thought baseline and public filtering.

F. Ablation on Temperature for REPEATED SAMPLING and IDEASEARCH

See Figure 25. We sweep over temperature increments of 0.1 from 0.0 to 1.2, inclusive, with top-p of 0.95, on REPEATED SAMPLING and IDEASEARCH.

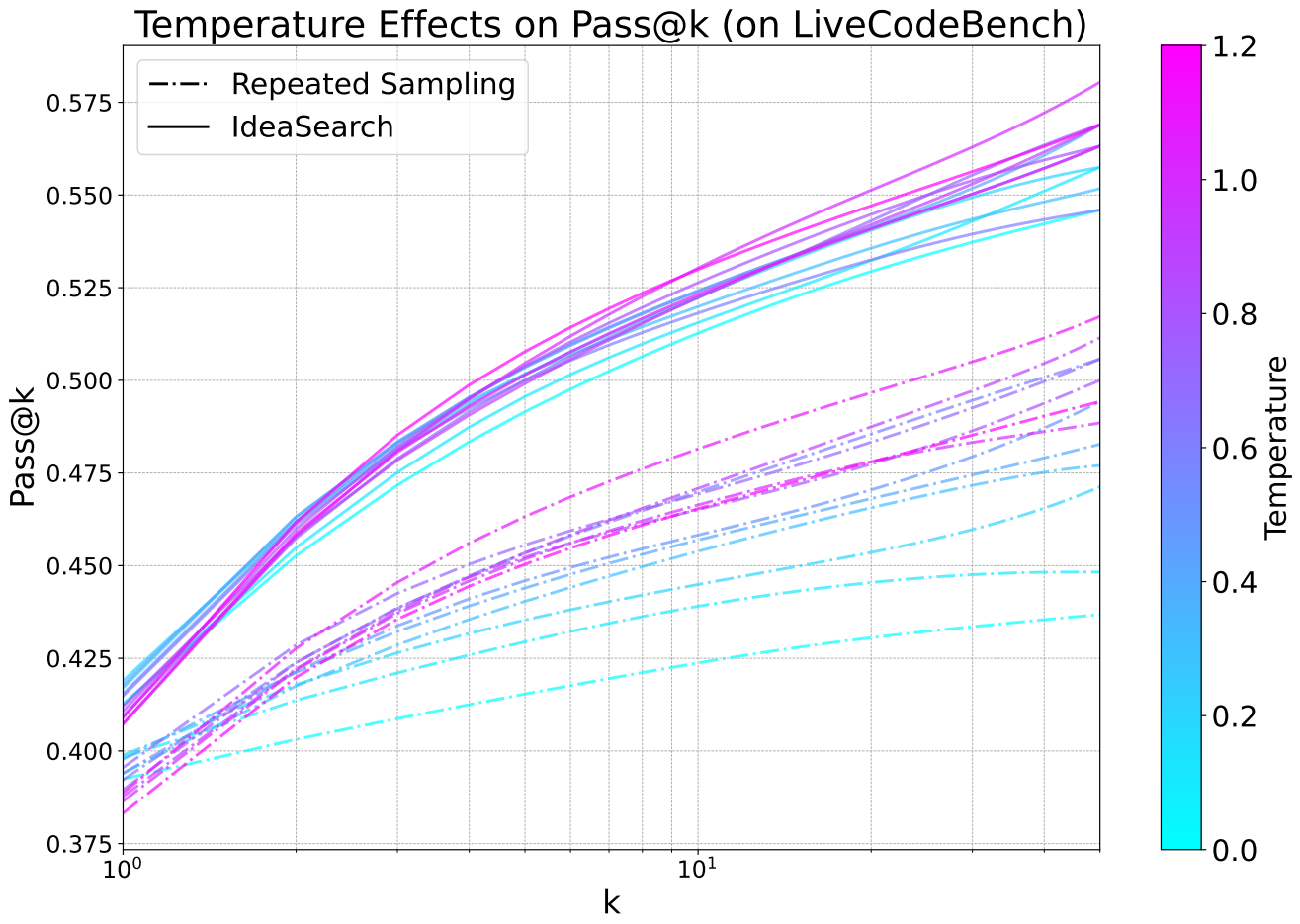


Figure 25: Sweep over temperature in 0.1 increments from 0.0 to 1.2. REPEATED SAMPLING and IDEASEARCH both exhibit pass@k improvements at higher temperature, although it seems that higher temperatures may begin to plateau.

G. Diversity Score vs Search Improvement Plots for MBPP+ and HumanEval+

See Figures 26, 27, 6. Each figure is made through running the diversity measure as described in Section 6.1 on the generated codes of each run, then compared with the relative gain from pass@k compared to pass@1.

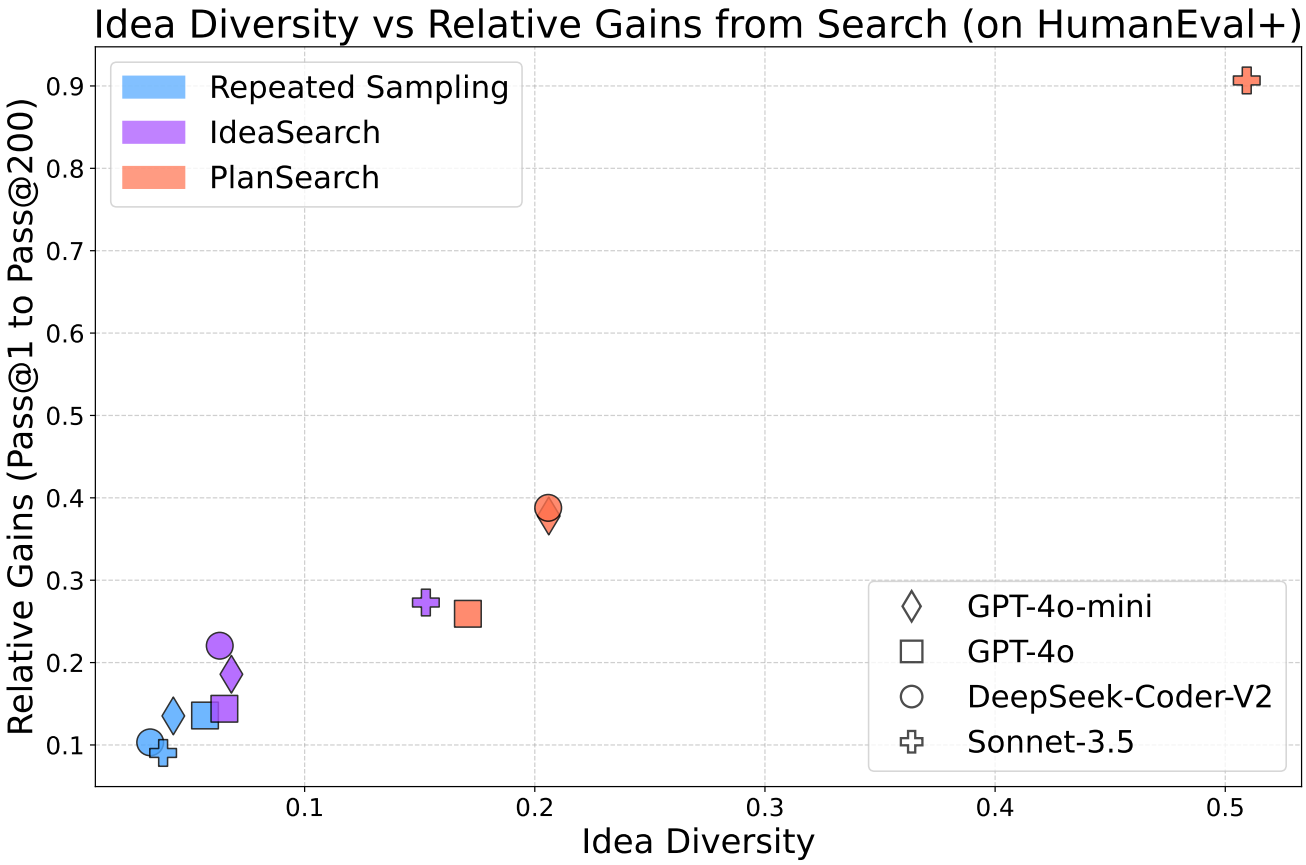


Figure 26: Relationship between the measured diversity score as described in Section 6.1 (where higher is more diverse) and relative improvement from the pass@1 of the method to the pass@200 of the method.

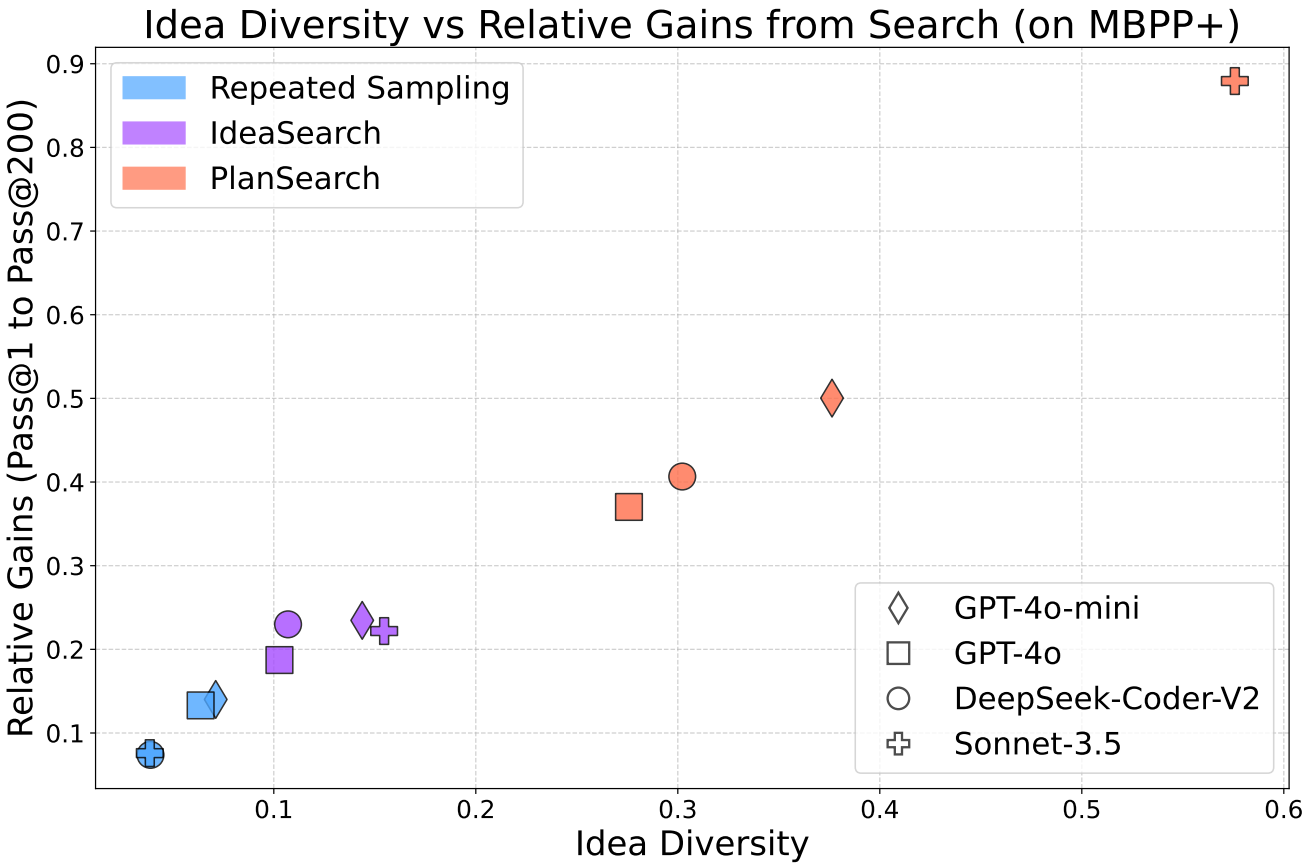


Figure 27: Relationship between the measured diversity score as described in Section 6.1 (where higher is more diverse) and relative improvement from the pass@1 of the method to the pass@200 of the method on MBPP+.

H. Base Models vs. Instruct Models for Large Samples

We find that base models, despite performing poorly relative to their instruct counterparts for evaluated with pass@1, will frequently match or even exceed performance on pass@k for sufficiently high k. This is likely due to higher amounts of diversity in base models, which have not undergone post-training designed to elicit a single strong response from the model.

We see this effect across all models for HumanEval+ and MBPP+, but only the DeepSeek-Coder-V2 family for LiveCodeBench.

See Figures 29, 30, 31 for Llama-3.1-8b pass@k comparisons.

See Figures 32, 33, 34 for Llama-3.1-70b pass@k comparisons.

See Figures 3, 35, 36 for DeepSeek-Coder-V2-Lite pass@k comparisons.

We also ran Llama-3.1-8b and DeepSeek-Coder-V2-Lite pass@k comparisons for k up to 10,000; see Figures 37, 28.

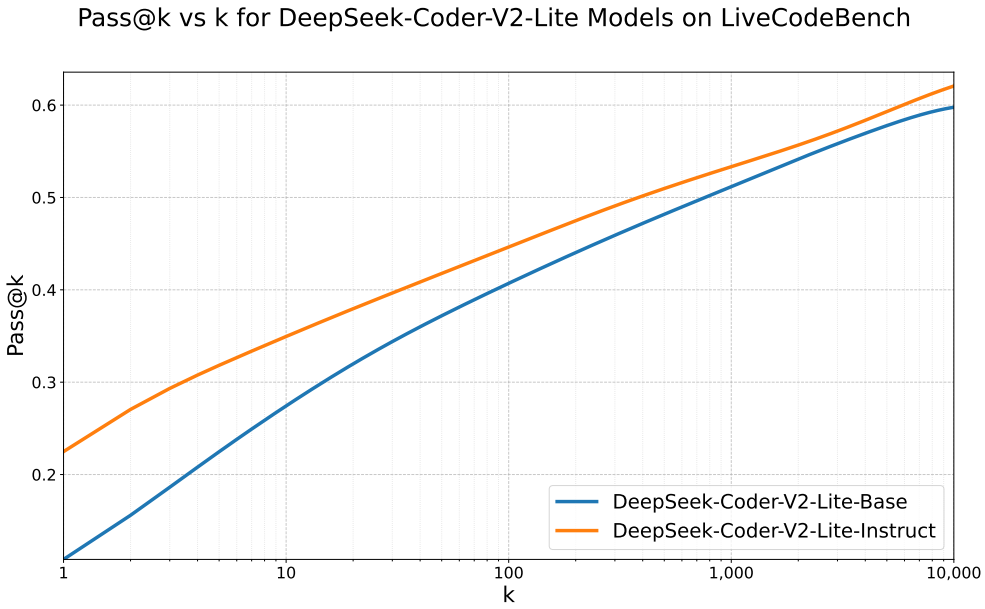


Figure 28: Pass@k curves comparing DeepSeek-Coder-V2-Lite's base and instruct versions on LiveCodeBench with up to 10,000 completions.

I. Base Models vs. Instruct Models with Public Test Filtering

We repeat the graphs from Appendix H, but with public test filtering. We find that base models with public test filtering almost always exceed the pass@1 of their instruct model variants.

See Figures 38, 39, 40 for Llama-3.1-8b pass@k comparisons with public test filtering.

See Figures 41, 42, 43 for Llama-3.1-70b pass@k comparisons with public test filtering.

See Figures 44, 45, 46 for DeepSeek-Coder-V2-Lite pass@k comparisons with public test filtering.

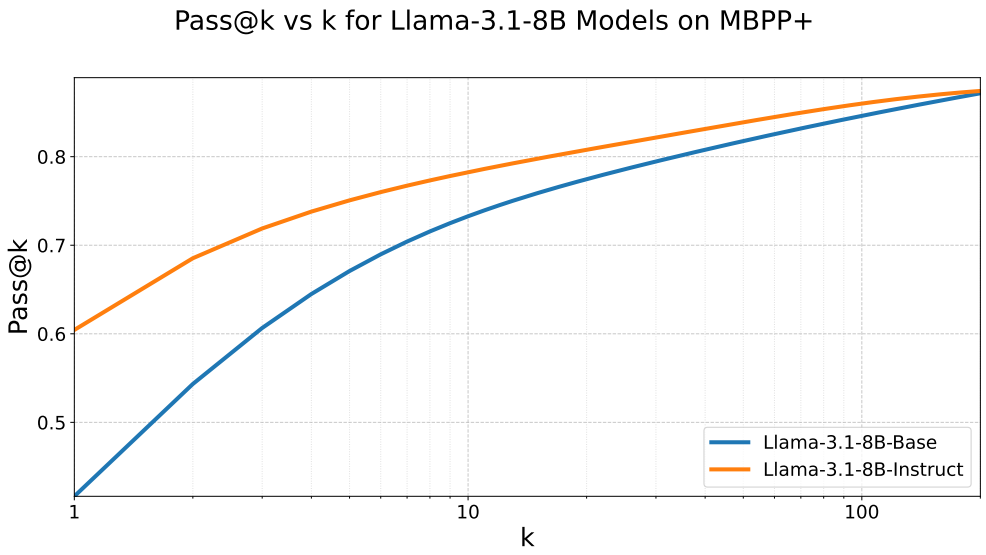


Figure 29: Pass@k curves comparing Llama-3.1-8B's base and instruct versions on MBPP+.

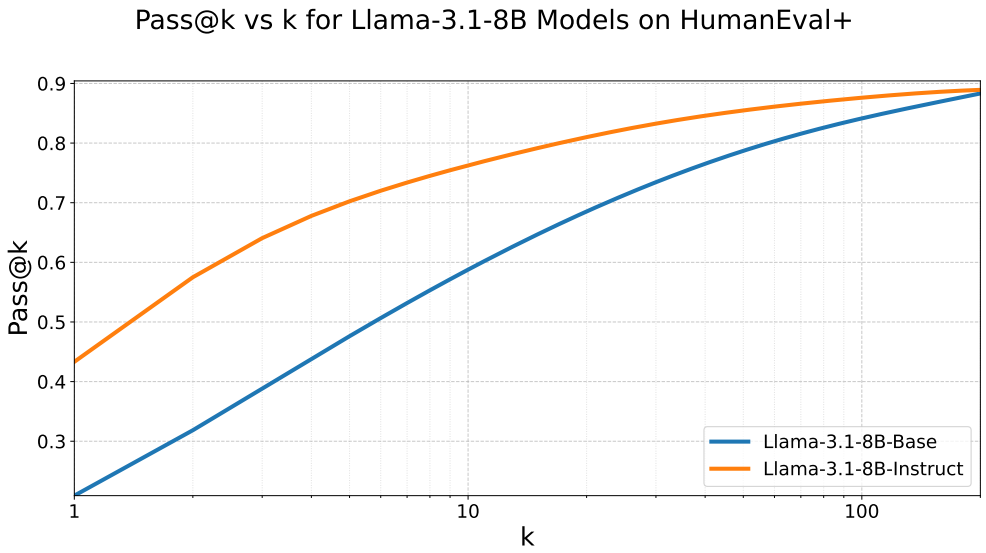


Figure 30: Pass@k curves comparing Llama-3.1-8B's base and instruct versions on HumanEval+.

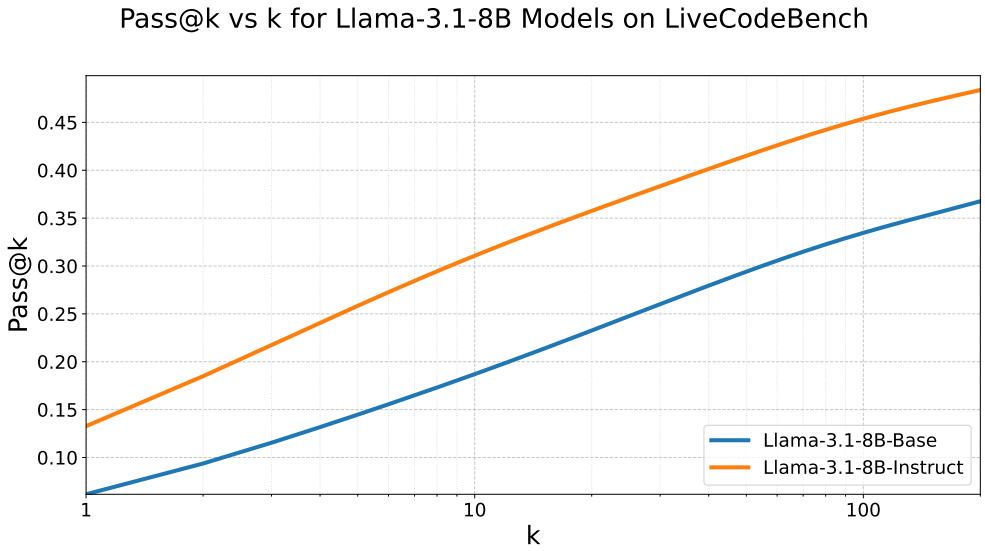


Figure 31: Pass@k curves comparing Llama-3.1-8B's base and instruct versions on LiveCodeBench.

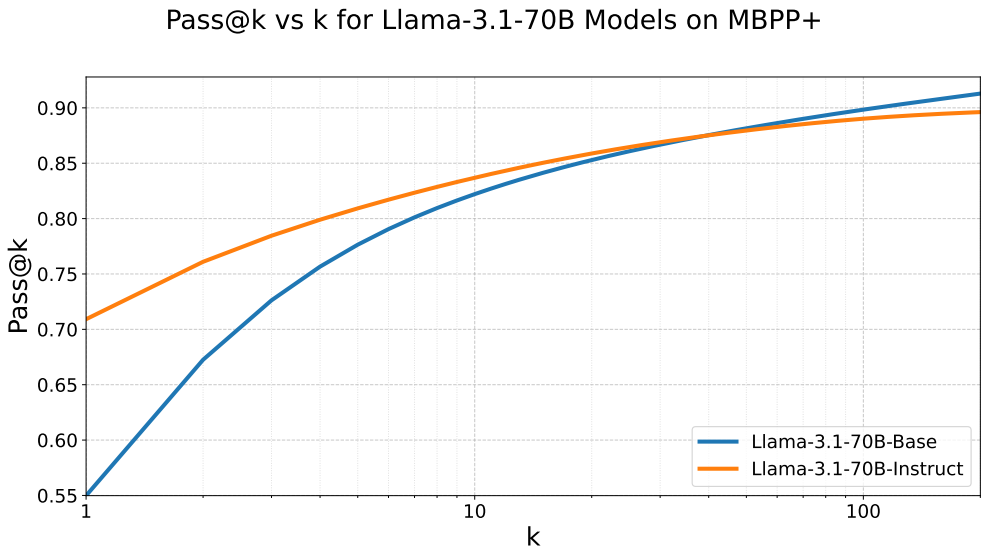


Figure 32: Pass@k curves comparing Llama-3.1-70B's base and instruct versions on MBPP+.

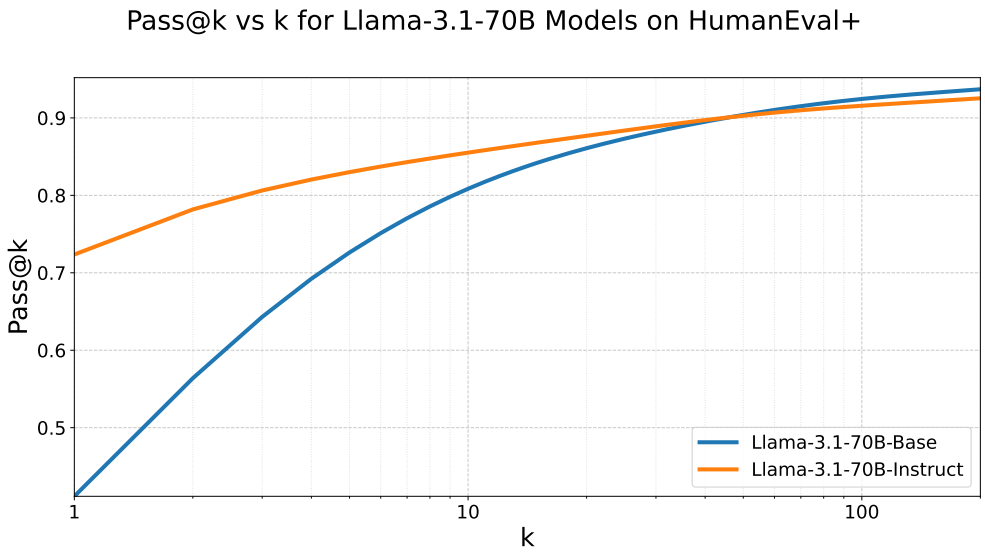


Figure 33: Pass@k curves comparing Llama-3.1-70B's base and instruct versions on HumanEval+.

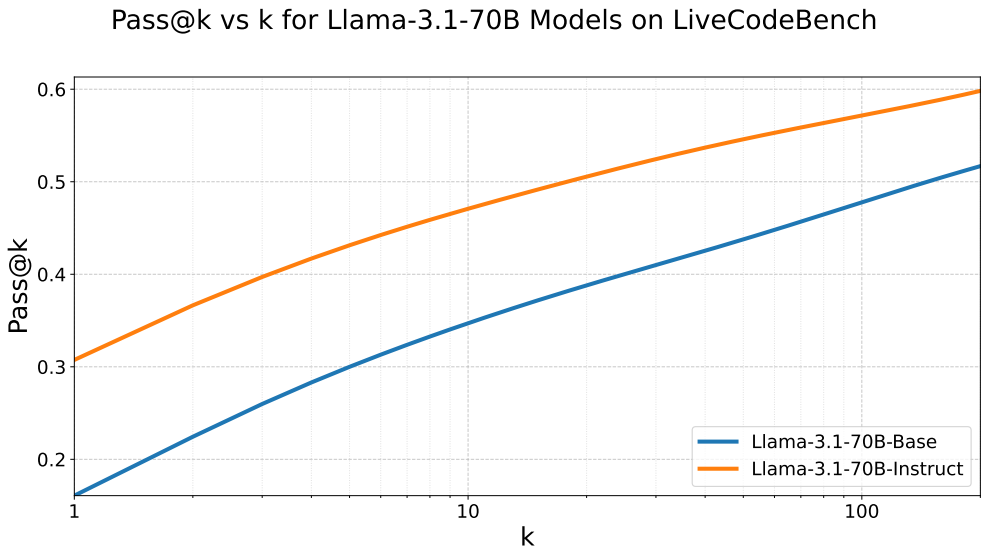


Figure 34: Pass@k curves comparing Llama-3.1-70B's base and instruct versions on LiveCodeBench.

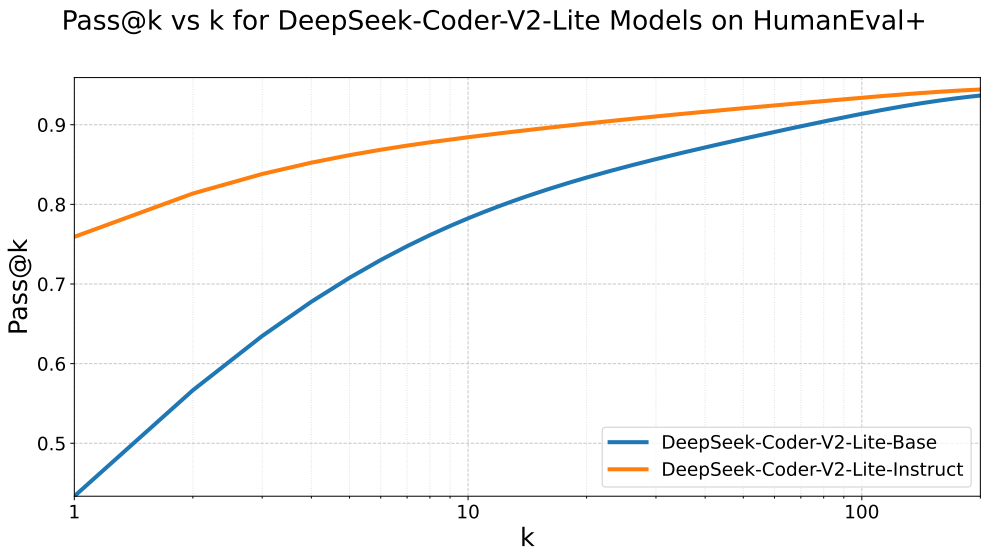


Figure 35: Pass@k curves comparing DeepSeek-Coder-V2-Lite's base and instruct versions on HumanEval+.

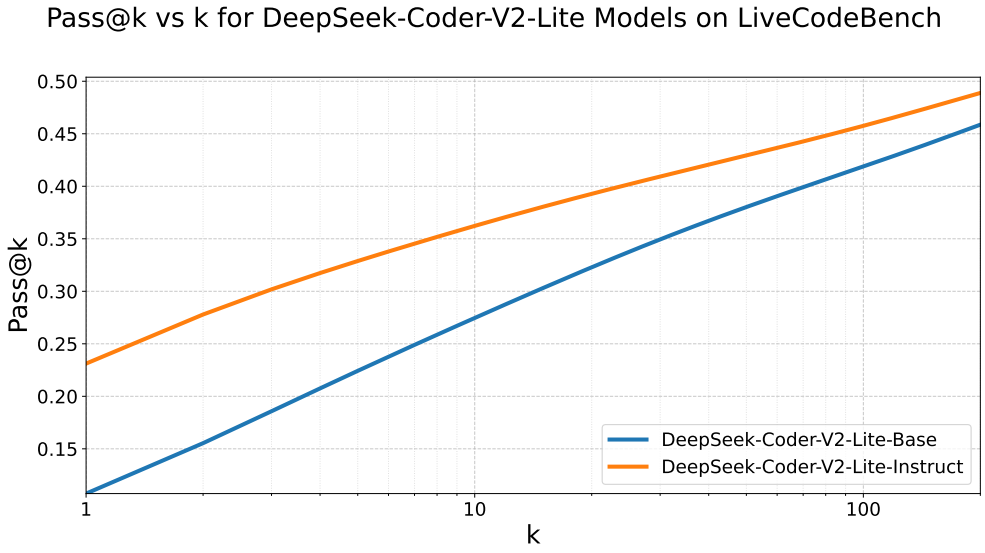


Figure 36: Pass@k curves comparing DeepSeek-Coder-V2-Lite's base and instruct versions on LiveCodeBench.

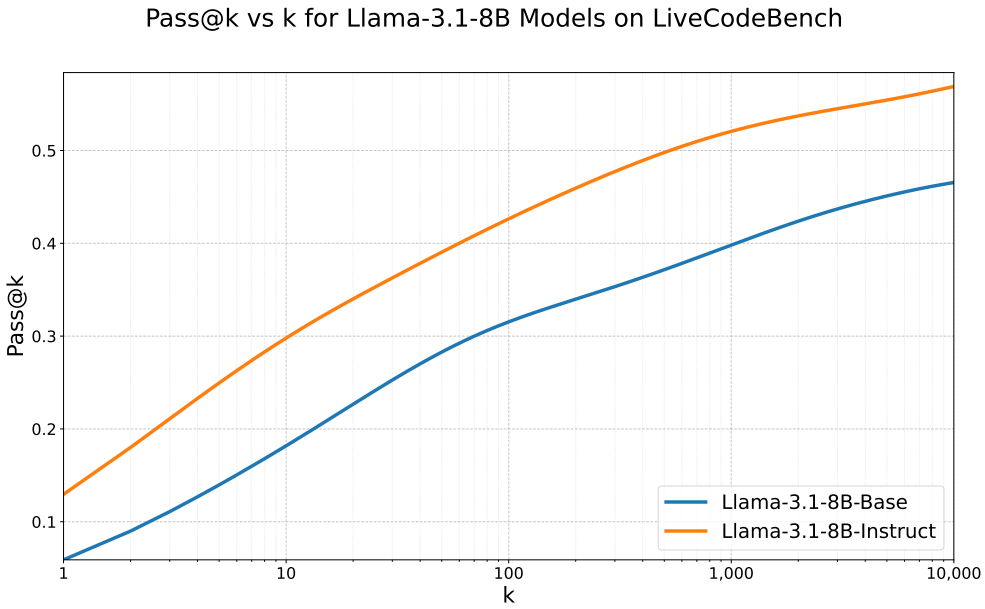


Figure 37: Pass@k curves comparing Llama-3.1-8B's base and instruct versions on LiveCodeBench with up to 10,000 completions.

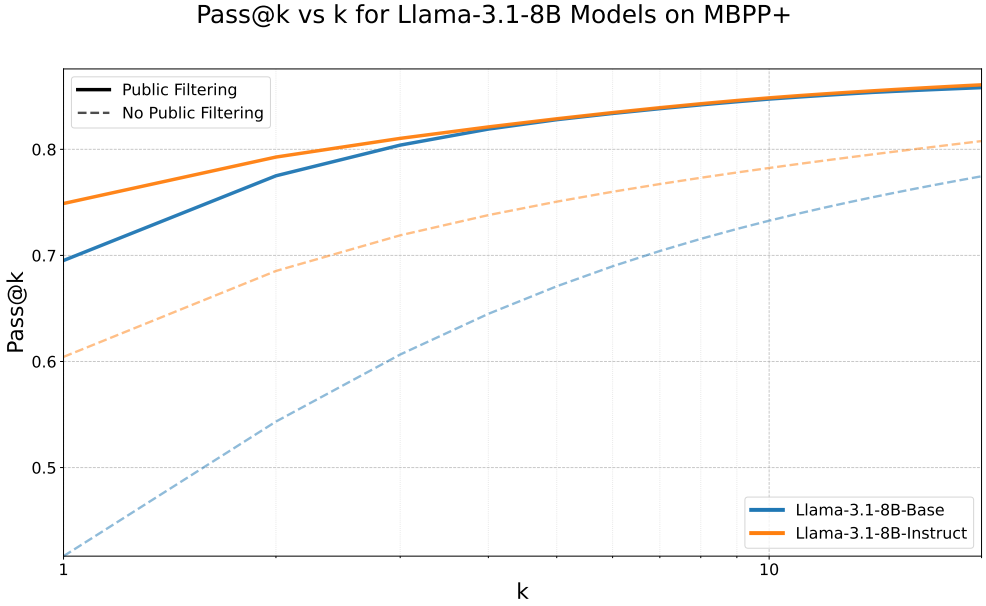


Figure 38: Pass@k curves comparing Llama-3.1-8B's base and instruct versions on MBPP+ with public test filtering.

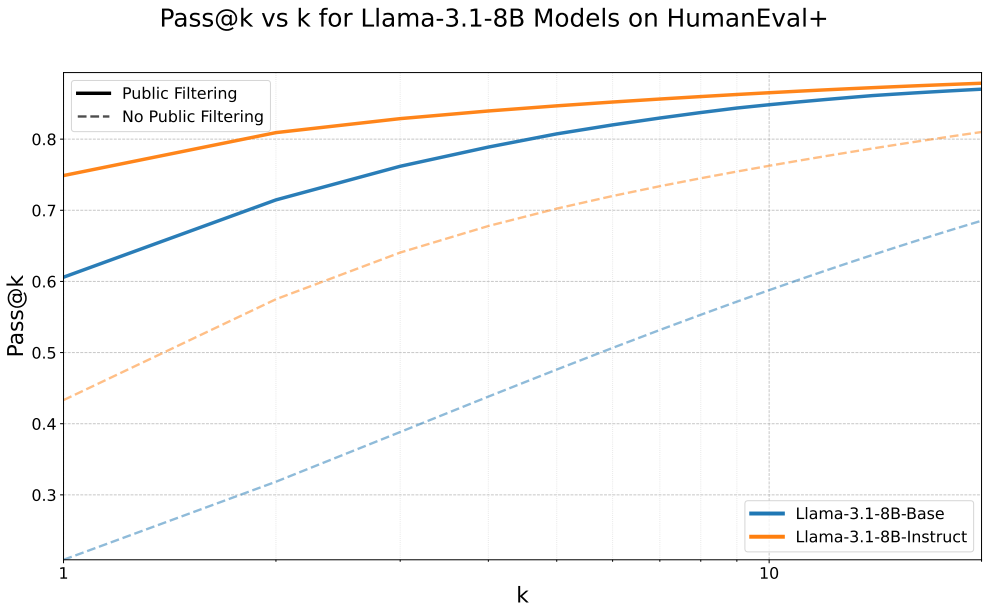


Figure 39: Pass@k curves comparing Llama-3.1-8B's base and instruct versions on HumanEval+ with public test filtering.

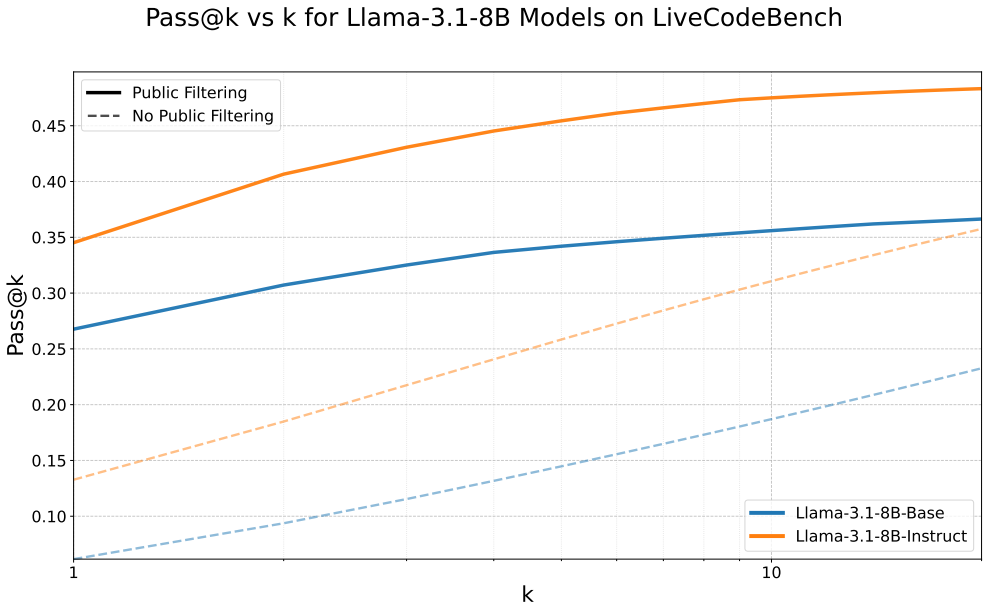


Figure 40: Pass@k curves comparing Llama-3.1-8B's base and instruct versions on LiveCodeBench with public test filtering.

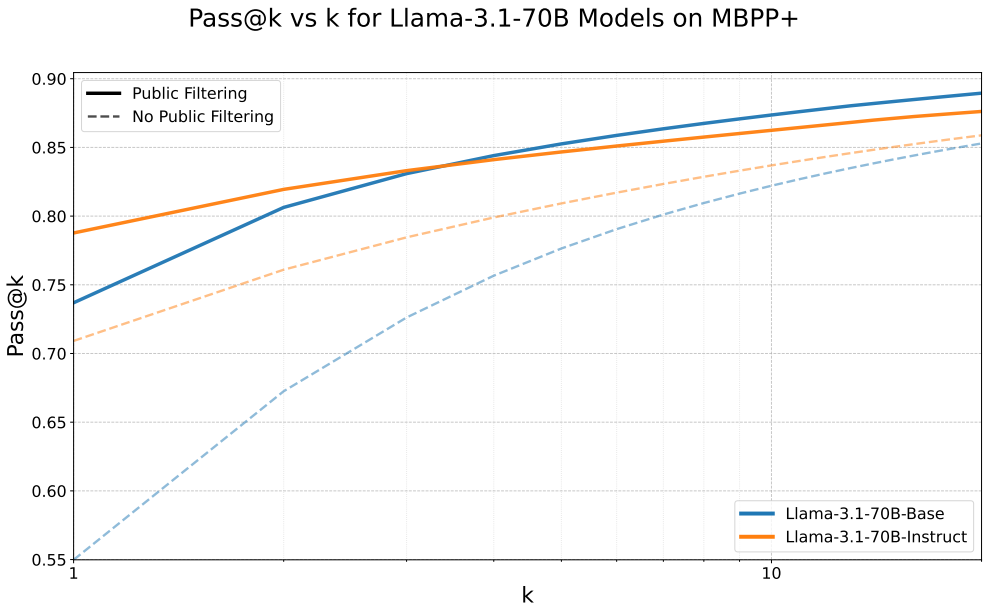


Figure 41: Pass@k curves comparing Llama-3.1-70B's base and instruct versions on MBPP+ with public test filtering.

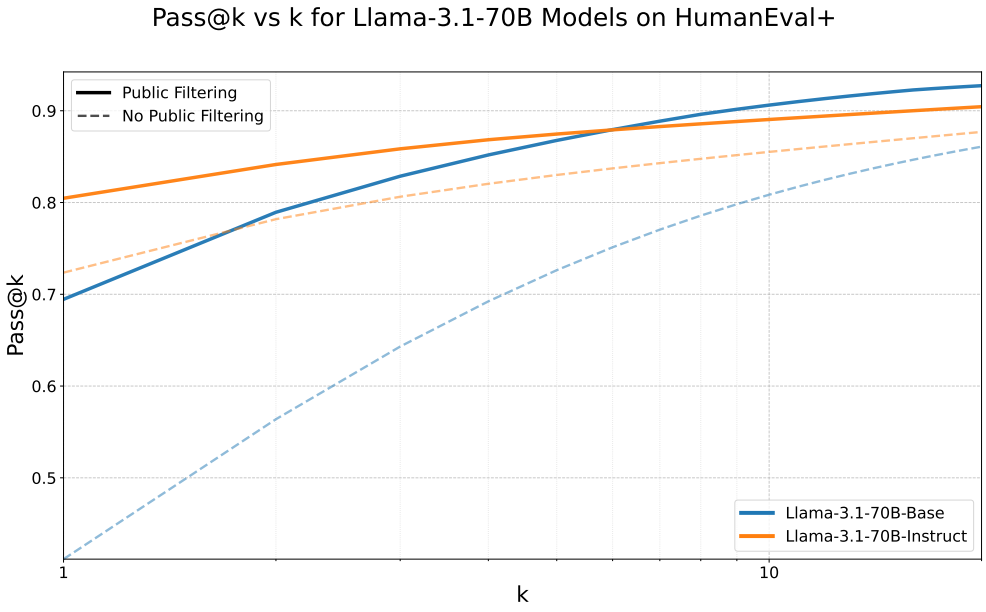


Figure 42: Pass@k curves comparing Llama-3.1-70B's base and instruct versions on HumanEval+ with public test filtering.

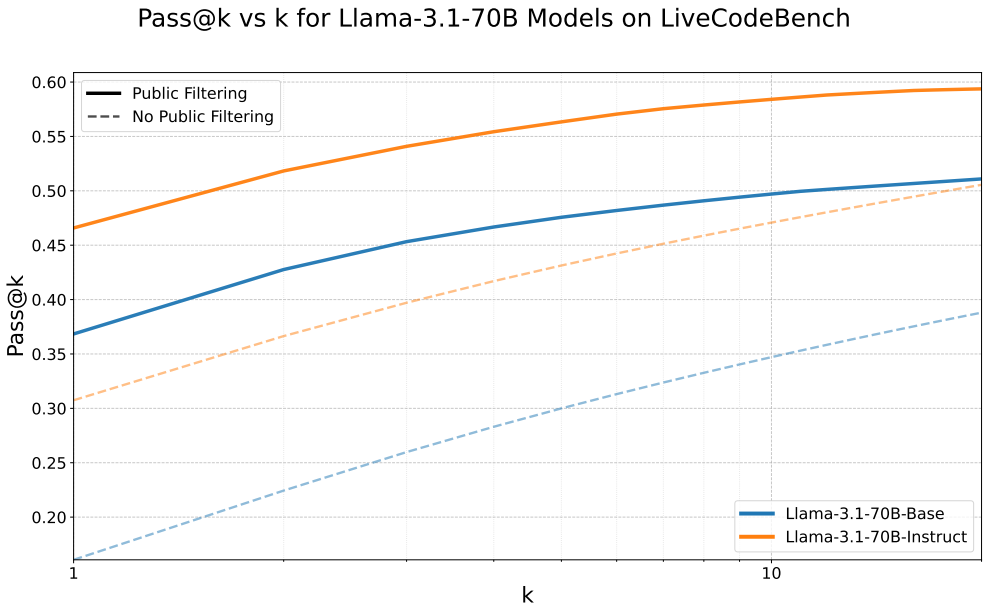


Figure 43: Pass@k curves comparing Llama-3.1-70B's base and instruct versions on LiveCodeBench with public test filtering.

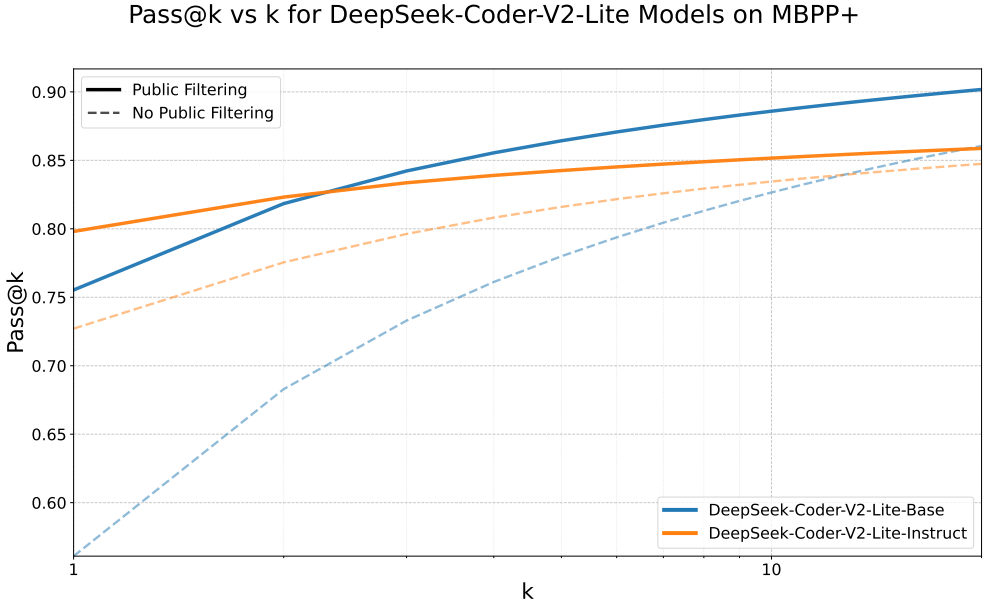


Figure 44: Pass@k curves comparing DeepSeek-Coder-V2-Lite's base and instruct versions on MBPP+ with public test filtering.

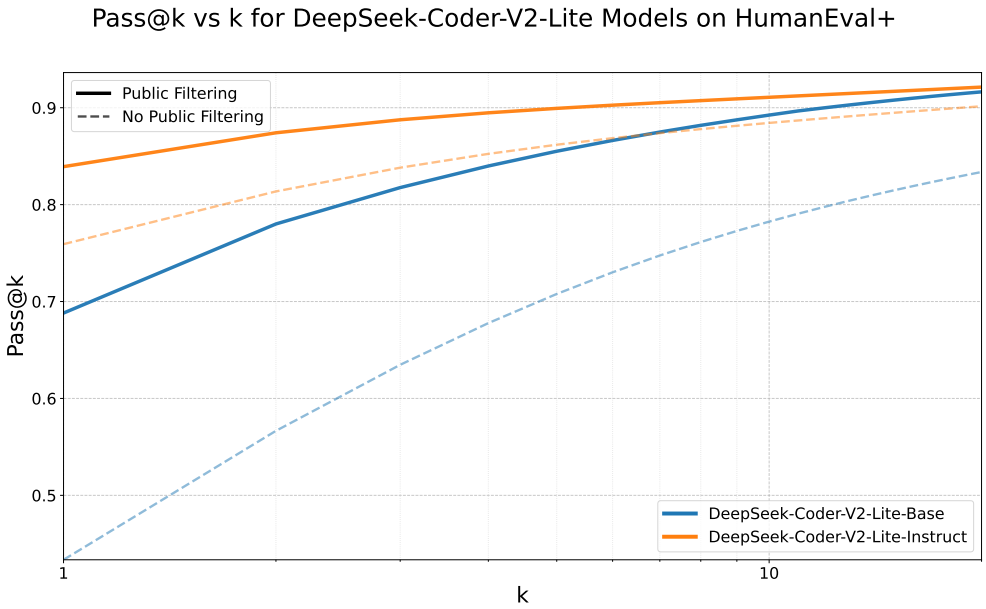


Figure 45: Pass@k curves comparing DeepSeek-Coder-V2-Lite's base and instruct versions on HumanEval+ with public test filtering.

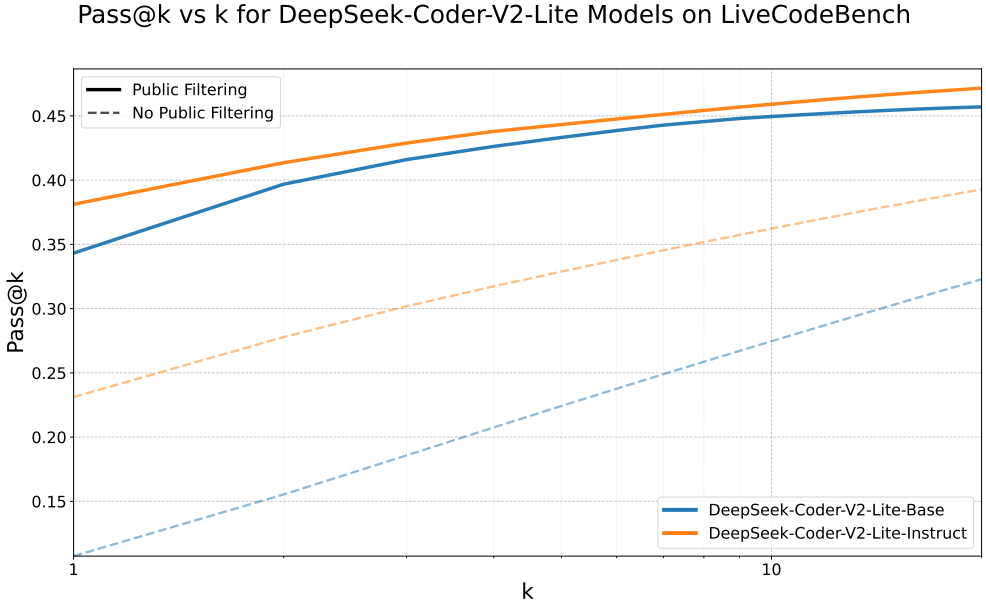


Figure 46: Pass@k curves comparing DeepSeek-Coder-V2-Lite's base and instruct versions on LiveCodeBench with public test filtering.

J. Prompts

J.1 Backtranslation

J.1.1 Backtranslate System Prompt

```
You are an expert Python programmer. You will be given an algorithmic question (problem
specification). You will return a high-level, natural language solution to the question,
like an editorial. You will NOT return any code. Be as creative as possible, going beyond
what you think is intuitively correct.
```

J.1.2 Implement Backtranslation Idea

```
You are an expert Python programmer. You will be given a question (problem specification)
and a natural language solution/tutorial that describes how to solve the problem. You will
generate a correct Python program that matches said specification and tutorial and passes
all tests. You will NOT return anything except for the program inside markdown codeblocks.
```

J.2 Repeated Sampling

```
You are an expert Python programmer. You will be given a question (problem specification)
and will generate a correct Python program that matches the specification and passes all
tests. You will NOT return anything except for the program inside Markdown codeblocks.
```

J.3 Simple Idea

```
You will given a competitive programming problem; please output a high-level description
of how to solve the problem in natural language. Below are examples:

Example input: PROBLEM DESCRIPTION HERE
Example output: EXAMPLE OUTPUT HERE
Here is the competitive programming problem: PROBLEM TO SOLVE

Brainstorm a high-level, natural language solution to the problem above. Note that your
intuition may lead you astray, so come up with simple, creative ideas that go beyond what
you would usually come up with and go beyond your narrow intuition. Brainstorming solutions
that do not seem intuitively correct IS CRUCIAL.
```

J.4 PlanSearch

J.4.1 Prompt for Observation Part 1

```
You are an expert Python programmer. You will be given an competitive programming question
(problem specification). You will return several useful, non-obvious, and correct observations
about the problem, like hints to solve the problem. You will NOT return any code. Be as
creative as possible, going beyond what you think is intuitively correct.
```

J.4.2 Prompt for Observation Part 2

```
You are an expert Python programmer. You will be given an competitive programming question
(problem specification) and several correct observations about the problem.

You will brainstorm several new, useful, and correct observations about the problem, derived
from the given observations. You will NOT return any code. Be as creative as possible, going
beyond what you think is intuitively correct.
```

J.4.3 Combining Observations

```
Here is a sample prompt from the function with placeholders:

Here is the competitive programming problem:

Problem statement placeholder

Here are the intelligent observations to help solve the problem:

Observation 1 placeholder
Observation 2 placeholder
Observation 3 placeholder

Use these observations above to brainstorm a natural language solution to the problem above.
Note that your intuition may lead you astray, so come up with simple, creative ideas that
go beyond what you would usually come up with and exceeds your narrow intuition.
Quote relevant parts of the observations EXACTLY before each step of the solution. QUOTING
IS CRUCIAL.
```

J.5 Measuring Diversity

```
You are an expert Python programmer. You will be given a competitive programming problem
and two pieces of code which are attempts to solve the problem. For your convenience, you
will also be given the idea for each code, summarized in natural language. You will be asked
to answer whether the ideas behind the code are the same. You must ONLY output 'Yes.' or
'No.'
```

K. Competitive Programming

Competitive programming is a popular subset of programming tasks that involve solving complex algorithmic reasoning. Typically, problems consist of a problem statement (written in natural language) P, with associated tests: $(x_i, y_i), i \in \{1, \ldots, m\}$, for which any solution must pass all of them.

The number of tests m depends on the problem, but typically ranges on the order of 25 to 100. A small subset of the tests are typically given to the solver (we call these public tests) to use as validation that their program passes simple cases. The rest of the tests are hidden. Solutions to the problems must generally pass all the tests to be considered correct. Formally, we let $f(x)$ denote the output of said code ran on input x. The solution code is considered correct (passing) if and only if $f(x_i) = y_i$ for all $i \in \{1, \ldots, m\}$.

Each dataset consists of many (on the order of low-hundreds) independent problems, and models are evaluated on each of these problems independently.

L. A Model of Repeated Sampling: Pass@k

Consider a simplified model of repeated sampling for code generation. Suppose we have a dataset $D = \{P_1, \ldots, P_l\}$ with l problems. For some problem P_i, define the probability p_i as the probability that our code generation model solves the problem P_i in one submission. The pass@k [13, 24] metric (for problem P_i) is defined as the probability that our code generation model solves the problem P_i at least once out of k submissions. Thus, if we know the true p_i of our model, we may compute our pass@k simply:

$$\text{pass@k}_i = 1 - (1 - p_i)^k \tag{2}$$

$$\text{pass@k} = \sum_i \text{pass@k}_i / l \tag{3}$$

However, it turns out that for $k > 1$, the naïve estimator as seen in Equation 2 is biased, if we sample $n_i \geq k$ from our code model to solve P_i, $c_i \leq n_i$ are correct, and compute $p_i = c_i / n_i$ [13]. Instead, pass@k_i is typically computed using the unbiased estimator:

$$\text{pass@k}_i = 1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \tag{4}$$

Note that reporting pass@k on a dataset where $l = 1$ is rather pointless, since pass@k can be derived using only pass@1_1 and n_1. Every curve, over a suitable range of k values, will look like the *S-curve* seen in Figure 47 (as k is plotted on a log scale).

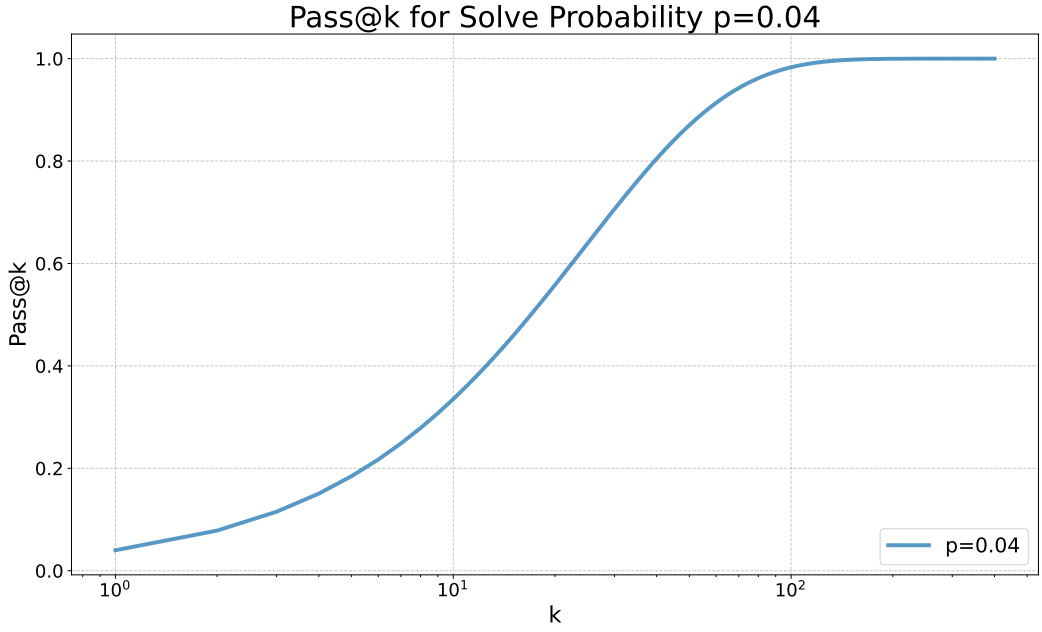


Figure 47: A simple pass@k '*S-curve*' plotted with $1 - (1 - p)^k$, where $p = 0.04$.

However, with datasets where $l > 1$, models are able to differentiate themselves through larger k, since the overall pass@k is an average of these l curves. For example, for $l = 3$, it is less optimal to have solved

probabilities of Set1 $= \{0.001, 0.7, 0.9\}$ versus Set2 $= \{0.05, 0.1, 0.25\}$, in the regime of roughly $k = 20$ to $k = 2,000$ (in which both converge to 1), even though Set1 has a pass@1 of 53% and Set2 has a pass@1 of 13%. See Figure 48.

Although not shown in the graph, Set2 converges close to 1 at roughly $k = 400$, several orders of magnitude below Set1. In addition, note that the slight notch seen in Set1's curve at large k is due to the presence of low, but non-zero solve-rates, which can be seen in empirical pass@k curves later on. (These can be thought as the beginning of the 'ramping-up' regime of the typical *S-curves* in Figure 47.)

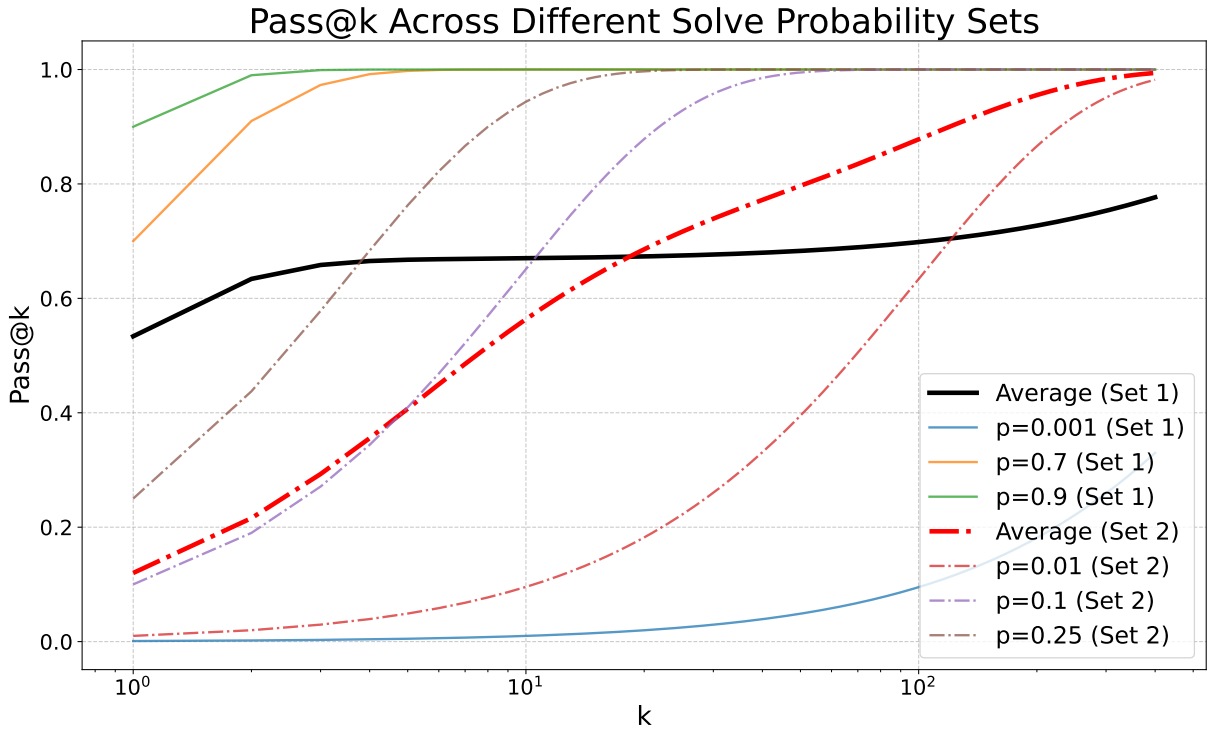


Figure 48: Two pass@k curves on a hypothetical dataset of length $l = 3$, and the solve probabilities of Set 1 are $\{0.001, 0.7, 0.9\}$ and Set 2 are $\{0.05, 0.1, 0.25\}$. Note that the pass@1 is 53% and 13%, respectively. However, at roughly $k = 20$, Set 2 surpasses Set 1 and within an order of magnitude, achieves pass@k of roughly 1.0.

M. Mathematics of the Diversity Measure

While our choice of a diversity metric is intuitive, one should note that there are a number of intriguing details that result from our definition. In particular, it is not necessarily the case that a model that outputs k unique ideas out of n samples to achieve a diversity score of $\frac{k}{n}$. Consider an example of $n = 9$ codes, separated into 3 cliques of 3, where each clique implements the same idea (and separate cliques implement separate ideas). In this setup, $\frac{1}{3}$ of ideas are unique, but in our metric, there are 3 matching idea pairs (and 9 total matching idea pairs) out of $\binom{9}{2} = 36$, for a diversity score of $1 - \frac{9}{36} = \frac{3}{4}$.

N. Biased Estimator for Pass@K Due to Non-Independence of PLANSEARCH

From a pure theoretical standpoint, the expression is biased (if using the same interpretation), but it still leads to a similar interpretation—computing the probability that a subset of size k drawn from the set of

samples we already generated contains at least one success. (These given samples were generated by one run of PLANSEARCH.) As such, in theory, the estimator may be slightly biased in the PLANSEARCH case when computing its true pass@k. In practice, we do not believe this to be a large concern, especially as our primary results feature a relatively large $k = 200$.