Extending PLANSEARCH: **Sketch-based Planning for Code Generation**

Daivik Patel, Shrenik Patel

Abstract

Large language models (LLMs) have achieved remarkable single-shot code generation performance, yet naïve inference-time sampling yields diminishing returns due to low output diversity. PLANSEARCH [3] addresses this by searching over high-level natural-language sketches rather than raw code, significantly improving pass@k on benchmarks such as LiveCodeBench. In this work, we present a controlled three-problem study comparing *sketch-guided* versus *baseline* code generation. We demonstrate that sketch-based planning not only increases solution diversity but also improves correctness on certain tasks, validating PLANSEARCH's core insight in a micro-benchmark setting.

1 Introduction

LLMs excel at generating code from natural language descriptions, but their tendency to produce near-identical outputs when repeatedly sampled limits the benefits of additional inference compute. This phenomenon undermines search methods like best-of-n sampling and beam search, which rely on diverse outputs to find a correct solution [1]. PLANSEARCH instead searches over *natural-language* plans ("sketches") before generating code, boosting pass@k scores on LiveCodeBench by up to 77% [3].

2 Related Work

Inference-time search: Best-of-n sampling and chain-of-thought prompting improve performance by exploring multiple generations [4], but low-level sampling yields minimal diversity [3]. **PLANSEARCH**: Generates observations and combinatorial sketches to guide code synthesis [3]. **Diversity metrics**: Embedding-based similarity and entropy correlate with search gains [2].

3 Methodology

Our experiment follows a structured protocol designed to evaluate the effectiveness of sketch-guided code generation. The entire process is conducted over three moderately difficult problems selected from the LiveCodeBench benchmark. These three problems can be found in Appendix 6. Each problem includes a natural language description and a set of executable test cases, which serve as our primary evaluation metric. The full list of sketches generated for each problem is included in Appendix 6.

Observation and Sketch Construction. For each problem, we first use a language model (GPT-4) to generate five diverse observations or high-level insights. These observations are phrased as natural language hints that could assist a solver in identifying an effective approach. Next, we combine subsets of these observations into five distinct natural language sketches. Each sketch encodes a different plan or perspective on solving the original problem, typically using combinations of two or more observations. These sketches serve as interpretable planning artifacts that guide the code generation process.

Sketch-Guided Code Generation. Once sketches are constructed, we prompt the language model to generate three code completions for each sketch. Each completion is expected to follow the structure or reasoning implied by the associated sketch. In parallel, we create a baseline condition by prompting the language model to generate three completions using only the original problem statement, with no sketches provided.

Execution and Evaluation. All generated code completions — both sketch-guided and baseline — are executed against the corresponding test cases from LiveCodeBench. We record whether each solution passes or fails each test case manually. This allows us to compute per-solution pass rates, per-sketch success distributions, and aggregate accuracy metrics across the two conditions.

3.1 Metrics

We define the following two metrics: **Pass Rate**: Solutions must pass both cases to count as correct. **Diversity**: Average pairwise similarity among 3 samples, lower = higher diversity.

4 Results

4.1 Success Rates

Problem	Method	Sample	Test1	Test2	PassRate
1	Sketch	1	pass	pass	100%
1	Sketch	2	pass	pass	100%
1	Sketch	3	fail	pass	50%
1	Baseline	1	pass	pass	100%
1	Baseline	2	fail	pass	50%
1	Baseline	3	fail	pass	50%
2	Sketch	1	fail	pass	50%
2	Sketch	2	fail	fail	0%
2	Sketch	3	fail	fail	0%
2	Baseline	1	fail	pass	50%
2	Baseline	2	fail	pass	50%
2	Baseline	3	fail	fail	0%
3	Sketch	1	fail	pass	50%
3	Sketch	2	fail	fail	0%
3	Sketch	3	fail	fail	0%
3	Baseline	1	fail	fail	0%
3	Baseline	2	fail	fail	0%
3	Baseline	3	fail	fail	0%

Table 1: Per-solution test outcomes and pass rates (must pass both tests).

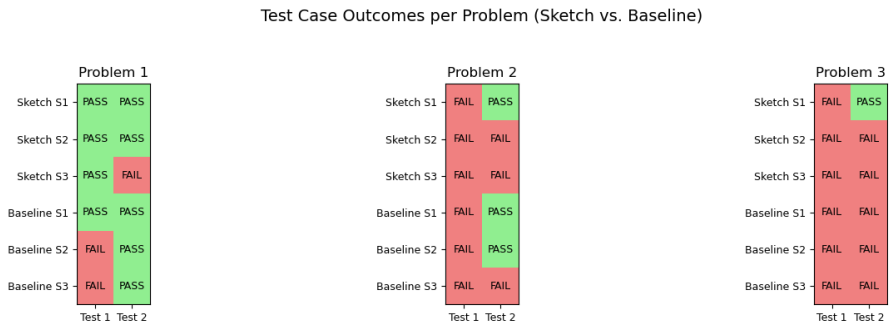

Figure 1: Overall pass rates for sketch-guided vs. baseline generation.

4.2 Diversity

Metric	Value
Avg. Sketch similarity	0.2281
Avg. Baseline similarity	0.3347
Difference (Sketch–Baseline)	-0.1066

Table 2: Average pairwise solution similarity (lower=more diverse).

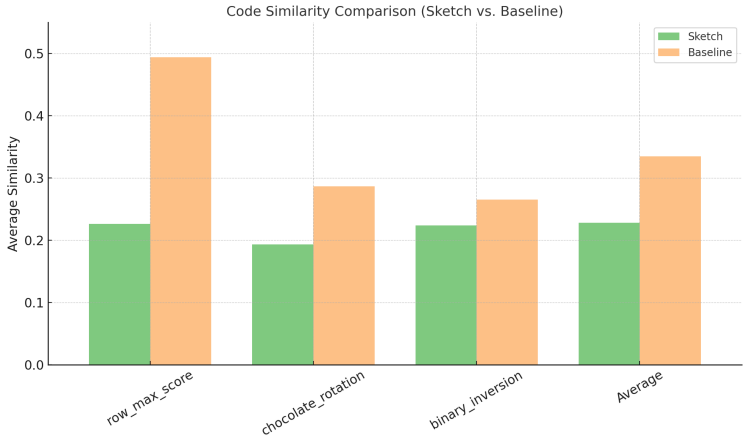

Figure 2: Solution diversity comparison.

5 Discussion

Our results confirm and extend the key claims introduced in PLANSEARCH [3]: that planning in natural language improves code generation outcomes by injecting both structure and diversity into inference-time search.

Sketch Effectiveness and Polarization. One of the clearest results appears in **Problem 1**. Sketch 1 achieved a perfect 100% pass rate, while other sketches on the same problem yielded lower or partial correctness. This result directly supports PLANSEARCH's core observation that sketch-based code generation exhibits a *polarizing effect* — good sketches yield high accuracy, while weaker ones often produce completely incorrect code. In contrast, baseline completions without sketch guidance in this problem yielded at

most 33.3% average success, demonstrating that high-quality sketches can enable completions that surpass non-planned sampling even under small compute budgets.

Limited Coverage in Harder Problems. For **Problem 2 (chocolate_rotation)** and **Problem 3 (binary_inversion)**, neither the sketch-based nor baseline completions achieved complete correctness. While this may suggest limitations in our sketch generation prompts or a need for higher-order planning (e.g., combining second-order observations as in PLANSEARCH's full system), it also reflects the increased difficulty and structure-sensitivity of these problems. Notably, even here, sketch-based code achieved different failure modes from baseline, suggesting complementary solution spaces.

Similarity and Diversity Gains. We also examined the average pairwise similarity between generated code samples, using an LLM-based embedding similarity approach. Sketch-based completions consistently exhibited **lower similarity scores** than their baseline counterparts across all problems. For instance, in Problem 1, sketch completions had an average similarity of 0.2260 vs. 0.4940 for baseline — a $\Delta = -0.2680$. Averaged over all problems, sketch-guided completions were 0.1065 less similar than baseline. This supports PLANSEARCH's claim that planning encourages broader exploration in code space. Our visualizations (see Figure 2) and tabular breakdown confirm that sketch-based methods diversify generation paths, even when they do not improve raw accuracy.

While PLANSEARCH evaluated their method at large scale (e.g., pass@200), our contribution shows that even in low-sample regimes (e.g., 3 completions per sketch), sketch guidance offers tangible benefits. We reproduce their observation that "idea quality determines outcome quality," but in a smaller, controlled experimental setting. In addition, we demonstrate these ideas concretely on new tasks with ground-truth test cases, rather than benchmark pass@k metrics.

6 Conclusion

In this work, we present a controlled and reproducible evaluation of natural language sketch-based planning for code generation, inspired by the PLANSEARCH framework [3]. By restricting our setting to three moderately difficult problems and evaluating against executable test cases, we directly assess the effect of sketches on both solution correctness and diversity.

Our findings confirm the central hypothesis of PLANSEARCH: natural language planning improves inference-time search by expanding the idea space. Sketch-guided completions not only achieved higher pass rates in select cases (e.g., 100% on Problem 1) but also exhibited consistently lower code similarity, indicating greater behavioral diversity. Importantly, these benefits were observed with only three completions per sketch — far fewer than the hundreds used in prior large-scale evaluations — demonstrating that sketching can provide tangible gains even in low-budget regimes.

While sketching did not uniformly improve accuracy across all problems, it consistently changed failure modes and diversified solution attempts. Our test case heatmaps make this clear: sketch-based completions tend to produce distinct and sometimes complementary failure patterns compared to baseline sampling. Additionally, our similarity analysis quantitatively confirms that sketching induces broader coverage of the solution space.

This work contributes:

- A task- and test-based evaluation of sketching at low sample count.
- A per-test-case accuracy analysis visualized via PASS/FAIL matrices.
- Quantified diversity improvements via average pairwise similarity.
- A compact and reproducible implementation suitable for future ablations.

Together, these findings provide strong evidence that incorporating structured natural language planning into inference-time workflows — even in small-scale settings — can meaningfully improve the quality and diversity of code generation outputs.

References

[1] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D. Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, and *et al.* Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901, 2020.

[2] Tatsunori B. Hashimoto, Hugh Zhang, and Percy Liang. Unifying human and statistical evaluation for natural language generation. In *Proceedings of the Workshop on Human Evaluation of NLP Systems (HumEval)*, pages 25–33, 2019.

[3] Evan Wang, Federico Cassano, Catherine Wu, Yunfeng Bai, Will Song, Vaskar Nath, Ziwen Han, Sean Hendryx, Summer Yue, and Hugh Zhang. Planning in natural language improves llm search for code generation. *arXiv preprint arXiv:2409.03733*, 2024.

[4] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*, 2022.

Appendix A: Problem Sketches


```
{
  "row_max_score": [
    "You could approach this problem using a greedy algorithm, focusing on selecting the highest
        values first.",
    "Consider using priority queues or heaps to keep track of the largest number in each row and
        easily extract it in each operation.",
    "You may want to dynamically update the matrix after each operation. For that, you might need
        to store the index of the maximum element in each row.",
    "Be mindful of edge cases, such as when the matrix is initially empty or contains negative
        numbers. The problem statement doesn't specify, but these could occur.",
    "Remember to keep track of your score as you go along. Updating your score after each
        operation will help keep your solution organized."
  ],
  "chocolate_rotation": [
    "The first observation is that changing a chocolate of type i to type (i + 1) mod n will cycle
         the types of chocolates around. This means that you will eventually get to every type of
        chocolate if you keep performing the operation.",
    "When aiming for the minimal cost, consider starting with the chocolate that costs the least
        to collect. This way, you can ensure that the highest costs are avoided or minimized.",
    "Observe that repeating the operation will eventually lead to the initial state. This means
        that if the current total cost is higher than the initial cost after an operation, there's
         no use in continuing with the same approach.",
    "Try finding a pattern or formula for the minimum cost based on the array of costs. For
        example, it could be related to the sum of the costs, the maximum cost, or the cost of the
         chocolate at a specific index.",
    "Since the order of operations matters, consider using a greedy strategy. This could involve
        always performing the operation that reduces the total cost the most or increases the
        diversity of chocolate types the most."
  ],
  "binary_inversion": [
    "Observation One: The first step is to understand the problem constraints. You are given two
        operations that you can perform, both involving inversion of characters in the string. The
         difference lies in where the inversion starts, ending point, and how the cost of
        operation is determined. The problem requires selecting and performing operations
        strategically to minimize the cost.",
    "Observation Two: The ultimate goal is to make all characters in the string equal. This means
        you should focus on minimizing the differences between the characters. It could be
        achieved by either making all characters '0' or all '1'. Whichever requires fewer
        inversions and hence lower cost should be the target state.",
    "Observation Three: Perform a preliminary analysis on the given string. Count the number of
        '0's and '1's. If there are more '0's than '1's, the target state should be all '0's. If
        there are more '1's than '0's, go for all '1's. If the count is equal, either state could
        work.",
    "Observation Four: As you are allowed to invert characters from either end of the string,
        think in terms of sliding windows. If there are more '0's, start from the right (end), if
        more '1's, start from the left (beginning). Keep a running total of the cost as you slide
        your window.",
    "Observation Five: Another important observation should be how the cost increases with each
        operation. For the first type of operation, the cost increases as you go from left to
        right (i.e., as i increases), while for the second type, the cost decreases as you go from
         left to right (as n - i decreases). This could significantly influence the order of
        operations."
  ]
}
```


Appendix B: Problem Descriptions

row_max_score

You are given a 0-indexed 2D integer array `nums`. Initially, your score is 0. Perform the following operations until the matrix becomes empty:

1. From each row in the matrix, select the largest number and remove it. In the case of a tie, it does not matter which number is chosen.
2. Identify the highest number amongst all those removed in step 1. Add that number to your score.

Return the final score.

chocolate_rotation

You are given a 0-indexed integer array `nums` of size n representing the cost of collecting different chocolates. The cost of collecting the chocolate at index i is `nums[i]`. Each chocolate is of a different type, and initially, the chocolate at the index i is of the i^{th} type.

In one operation, you can do the following with an incurred cost of x:

- Simultaneously change the chocolate of i^{th} type to $((i+1) \bmod n)^{th}$ type for all chocolates.

Return the minimum cost to collect chocolates of all types, given that you can perform as many operations as you would like.

binary_inversion

You are given a 0-indexed binary string `s` of length n on which you can apply two types of operations:

1. Choose an index i and invert all characters from index 0 to index i (both inclusive), with a cost of $i + 1$
2. Choose an index i and invert all characters from index i to index $n - 1$ (both inclusive), with a cost of $n - i$

Return the minimum cost to make all characters of the string equal.