MUTAGREP: Execution-Free Repository-Grounded Plan Search for Code-Use

Zaid Khan [1] **Ali Farhadi** [2] **Ranjay Krishna** [2] **Luca Weihs** [3] **Mohit Bansal** [1] **Tanmay Gupta** [2]

Abstract

When a human requests an LLM to complete a coding task using functionality from a large code repository, how do we provide context from the repo to the LLM? One approach is to add the entire repo to the LLM's context window. However, most tasks involve only fraction of symbols from a repo, longer contexts are detrimental to the LLM's reasoning abilities (Kuratov et al., 2024), and context windows are not unlimited. Alternatively, we could emulate the human ability to navigate a large repo, pick out the right functionality, and form a plan to solve the task. We propose MUTAGREP (**Muta**tion-guided **G**rounded **Re**pository **P**lan Search), an approach to search for plans that decompose a user request into natural language steps grounded in the codebase. MUTAGREP performs neural tree search in plan space, exploring by mutating plans and using a symbol retriever for grounding. On the challenging LongCodeArena benchmark, our plans use less than 5% of the 128K context window for GPT-4o but rival the coding performance of GPT-4o with a context window filled with the repo. Plans produced by MUTAGREP allow Qwen 2.5 Coder 32B and 72B to match the performance of GPT-4o with full repo context and enable progress on the hardest LongCodeArena tasks. Project page: zaidkhan.me/MutaGReP

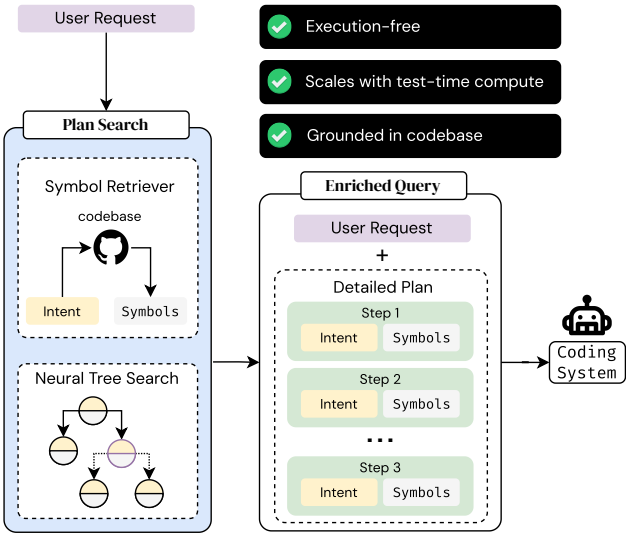

Figure 1. **MUTAGREP Overview** Given a user request that requires writing code against a specific codebase, we search for realizable plans to solve the user's request using LLM-guided tree search. Our search procedure uses a symbol retriever to constrain search to plans which are implementable with symbols available in the codebase and explores the search space by mutating plans. Each step of the plan consists of a natural language intent and symbols from the codebase that can be used to implement the intent. The user request along with the detailed plan serves as an enriched query that provides necessary context from the codebase to any downstream coding system to convert the plan to code. Our plan search benefits from test-time compute scaling and produces repo-grounded plans without requiring code execution.

1. Introduction

Code generation systems powered by LLMs are routinely tasked with writing new code given an existing large codebase. One approach to conditioning an LLM's generation on a repository is to utilize its working memory by concatenating all files in the codebase into a massive prompt. This is an inefficient use of finite context, because many programming tasks require only a small fraction of all symbols (functions, classes, global variables etc) in the codebase. Recent investigation also shows that leading LLMs are effective at utilizing less than 20% of their context lengths with a sharp decline in performance with increasing reasoning complexity (Kuratov et al., 2024). Can we do better?

Human programmers are able to understand complex codebases with a much smaller biological working memory. They achieve this by decomposing the target task into smaller steps and then using code search tools to identify relevant symbols (functions, classes etc) to use in each step. This is often an iterative process where the programmer interacts with the codebase to develop a realizable plan based on the symbols available in the codebase. The realizable plan can then be implemented.

[1]University of North Carolina, Chapel Hill [2]Allen Institute for Artificial Intelligence (Ai2) [3]Vercept AI (work done while at Ai2). Correspondence to: Zaid Khan <zaidkhan@cs.unc.edu>, Tanmay Gupta <tanmayg@allenai.org>.

Figure 2. **A repo-grounded plan created by MUTAGREP on a query from LongCodeArena**. Each plan step consists of a natural language intent with top-5 symbols retrieved from the codebase that might be useful for implementing the step.

In this work, we aim to replicate the ability of human programmers to iteratively search for a repo-grounded realizable plan to solve a user query given a codebase. This plan should wrap all relevant information from the codebase into a self-contained prompt which is human readable, editable and can be handed off to any code generation LLM or coding assistant for translation into code.

Specifically, given a codebase and a user query that requires writing code using symbols in the codebase, we aim to find a repo-grounded plan with multiple steps. Each step consists of a natural language intent and the symbols from the codebase that may be used to implement or *realize* the plan. A good plan is complete, concise, and faithful to the original query. These plans along with the original user query can be viewed as an enriched query that provides all relevant context from the codebase with detailed steps on how to solve the user query.

Since the space of all plans is semi-structured and massive, MUTAGREP formulates repo-grounded plan search as an LLM-guided tree-search problem. Each node in the tree represents a plan. Each step of the search process involves identifying the most promising node to expand and create children or successors of the node by mutating the plan. The mutation aims to make the successors more accurate, repo-grounded and realizable. When the search budget is exhausted the best plan is returned to the user. *Importantly,* MUTAGREP *does not require executing any code.*

The design space of MUTAGREP consists of four key components - successor function to mutate plans, a symbol retriever to ground intents to symbols in the codebase, a tree-traversal algorithm that decides the order in which to expand the nodes, and a plan ranker to select the most promising node to expand or to identify the best plan from the available nodes. We use the challenging LongCodeArena benchmark (Bogomolov et al., 2024) to thoroughly explore the design space of repo-grounded plan-search.

Our contributions include: (i) demonstrating the utility of repo-grounded plans for code-use (Table 2, Figure 7, Figure 8); (ii) formulating execution-free repo-grounded planning as LLM-guided tree search using an intent to symbol grounding function (Sec. 3); (iii) elucidating and studying the design space of repo-grounded plan search (Figure 5, Figure 6, Table 3); and (iv) demonstrating that plan search allows code-use to benefit from gains by scaling test-time compute (Figure 5 and Figure 6).

2. Related Work

Repository-grounded code generation. Existing work on repo-level code generation has explored two distinct directions. One line of work focuses on building software engineering agents that can edit real-world codebases to solve

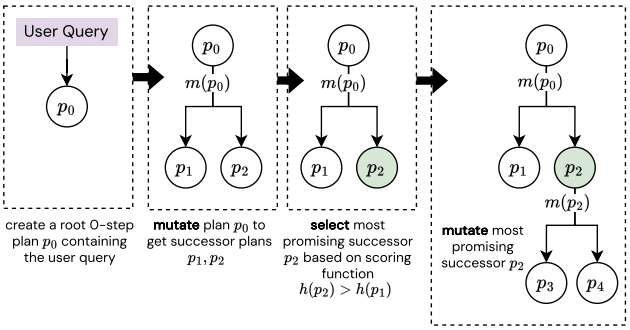

Figure 3. **Overview of plan search**. Each node in the tree is a repo-grounded plan. At every time step, a node is chosen for growing the tree and successors are created by mutating the chosen plan. We use an LLM to implement the successor function.

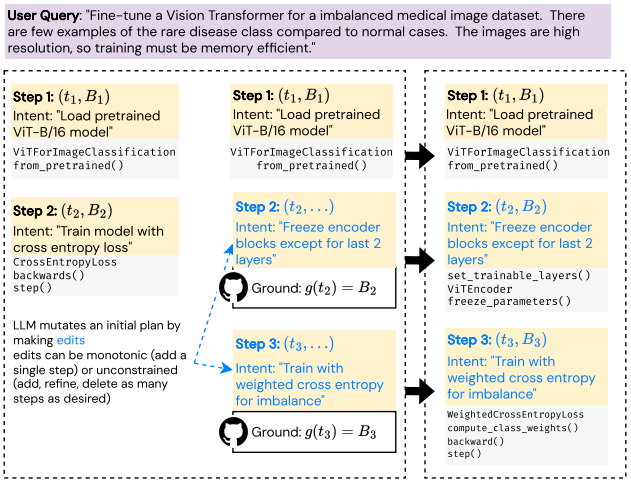

Figure 4. **Mutation and grounding.** The successor function s mutates a plan (left-most column) to generate new plans (right-most column). For each modified intent (t_2 and t_3), the grounding function g maps the intent to symbols that might be used to implement the intent (B_2 and B_3).

Github issues. The challenge here involves understanding multiple files and coordinating edits across those files while executing unit tests to validate these changes. A popular benchmark for this paradigm is SWE-bench (Jimenez et al., 2024) with several systems inching towards the performance of human engineers (Yang et al., 2024; Xia et al., 2024).

Code-use is an alternate paradigm that involves using a codebase as a library to write new code to solve a user's query. This requires the code generation system to discover the relevant symbols (functions, classes, variables etc.) in the codebase, understand the syntax and function of these symbols, and write code using these symbols to solve the task. CodeNav (Gupta et al., 2024) is a code-use agent that iteratively interacts with a keyword-based retrieval environment and an execution environment to solve the user's query. CodeNav repurposed tool-use benchmarks (Wang et al., 2024b; Ma et al., 2024; Li et al., 2023) to evaluate code-use by providing the agent with the codebase implementing the tools instead of tool prompts. However, given the limited number of tools, and the simplicity of tools (simple functions) and user queries, these repurposed benchmarks fail to test the LLMs on the challenges of real-world code-use.

In this work, we focus on the code-use scenario while using the recently released LongCodeArena (LCA) (Bogomolov et al., 2024) benchmark. Specifically, we use the library-based code generation challenge in the LCA benchmark suite which curates tasks using example scripts found in prominent Github repositories. Since these examples scripts are provided by the library authors to demonstrate using their library for real tasks, LCA tasks present a significantly more challenging and realistic test-bed for code-use than previous code-use evaluations.

Plan search for code generation. Recent work has shown the benefits of plan search for competitive programming tasks that require general knowledge of a programming language and its primitives. PlanSearch (Wang et al., 2024a) demonstrates that searching over natural language plans before generating code leads to more diverse solutions and better performance. While both PlanSearch and MUTAGREP requires searching for plans that decompose a user query into a sequence of simpler steps, we further need to constrain search to the space of realizable plans i.e. plans where all steps can be implemented using the target codebase.

Test-time search for code generation. Several systems such as AlphaCode(Li et al., 2022), CodeTree(Li et al., 2024), and CodeMonkeys(Ehrlich et al., 2025; Brown et al., 2024) have demonstrated impressive performance on code generation tasks by scaling test-time compute to search in the space of programs while using execution feedback to guide the search. Similar to code search, AlphaGeometry(Trinh et al., 2024) utilizes neural-guided tree spearch to explore the solution space of geometric theorems represented using a formal geometric language with rich verifiers for validating each step. While our work also uses search for code generation, we search in the space of repo-grounded plans without execution feedback or formal verifiers. Nonetheless, we show that plans produced by our approach provide necessary context from the target codebase for the task of repo-grounded code generation.

3. Method

Overview We formulate repository-grounded planning as a search problem over the space $\mathcal{P}$ of possible plans. Given the large space of possible plans and the semi-structured nature of plans, we employ a tree-based search algorithm

Table 1. Notation used throughout this paper

Notation	Meaning
$\mathcal{B}$	Set of all symbols (functions, classes, methods) in codebase
$\mathcal{L}$	Set of all finite character strings (natural language)
$t \in \mathcal{L}$	Natural language intent for a plan step
$B \subseteq \mathcal{B}$	Set of relevant symbols for implementing a step
$x = (t, B)$	Plan step: a tuple of intent and relevant symbols
$p = [x_1, ..., x_n]$	Plan: sequence of plan steps
$\mathcal{P}$	Space of all possible plans
$g : \mathcal{L} \to \mathbb{P}(\mathcal{B})$	Maps intents to relevant symbols where $\mathbb{P}$ is the power set
$s : \mathcal{P} \to \mathbb{P}(\mathcal{P})$	Generates set of possible next plans or successors
$h : \mathcal{P} \to \mathbb{R}$	Scoring function for ranking plans

using an LLM to guide the search process (Figure 3). Nodes in the tree represent the set of candidate plans explored so far with children created from parents via mutation using a successor function $s : \mathcal{P} \to \mathbb{P}(\mathcal{P})$ where $\mathbb{P}(\mathcal{P})$ denotes the power set of $\mathcal{P}$. The search process begins at the root node which consists of the high-level user query as the initial plan $p_0 \in \mathcal{P}$. At each step, a node is selected for mutation using the scoring function $h : \mathcal{P} \to \mathbb{R}$ (or using node expansion order as in depth-first search) and the successors are added to the set of candidates to be considered for future expansions. The process is repeated up to a user specified budget (nodes expanded), and the best plan (identified using h) is returned.

While this search procedure is applicable to a wide range of planning problems, we focus on choosing the space of plans, and appropriate mutation and scoring functions for the application of planning for repo-conditioned code generation. We conceptualize a plan $p \in \mathcal{P}$ as a sequence of steps $[x_1, ..., x_n]$ where each step x_i is a tuple (t, B) consisting of a natural language intent $t \in \mathcal{L}$ and a set $B \subseteq \mathcal{B}$ of symbols relevant to implementing the intent chosen from the set $\mathcal{B}$ of all symbols found in the codebase. To obtain relevant symbols, we construct a grounding function $g : \mathcal{L} \to \mathbb{P}(\mathcal{B})$ that maps natural language intents to relevant symbols. Having relevant symbols from the target codebase as part of the plan is what makes these plans repo-grounded and realizable.

For the successor function $s : \mathcal{P} \to \mathbb{P}(\mathcal{P})$, we explore two variants: an incremental version that only adds a new step to an existing plan, and an unconstrained version that can modify any part of the plan. Both variants use a language model to propose modifications to intents while using the grounding function to map intents to relevant symbols in the codebase.

Finally, we consider two scoring functions for ranking nodes: a heuristic function that encourages symbol diversity, and an LLM-based scoring function. For uninformed search like depth-first search, the scoring function is used solely for selecting the best plan after search completion, while for informed search like best-first search, the scoring function is also used to select the best node for expansion.

3.1. Successor Function and Grounding

The successor function $s : \mathcal{P} \to \mathbb{P}(\mathcal{P})$ determines how we explore the space of possible plans. For a given plan $p = [x_1, ..., x_n]$ where each $x_i = (t_i, B_i)$, the successor function must generate new plans while attempting to ensure each step remains grounded in symbols $\mathcal{B}$ available in the codebase.

3.1.1. SUCCESSOR FUNCTION VARIANTS

We consider two choices of the successor function:

Monotonic. The monotonic successor function s_m preserves all steps in the parent plan, only mutating the plan by adding new steps. Formally, given a plan $p = [(t_1, B_1), ..., (t_n, B_n)]$, $s_m(p)$ generates plans of the form $[(t_1, B_1), ..., (t_n, B_n), (t_{n+1}, B_{n+1})]$ where t_{n+1} is a new intent and $B_{n+1} \subseteq \mathcal{B}$ contains the symbols needed to implement it. This ensures the search progressively builds longer plans while maintaining previously discovered steps, evocative of monotonic relaxations in planning (Bonet & Geffner, 2001; McDermott, 1999; Hoffmann & Nebel, 2001).

Unconstrained. An unconstrained successor function s_u may perform arbitrary modifications to any part of the plan. For a plan p, $s_u(p)$ can generate plans with modified, deleted, or reordered steps, while maintaining the requirement that each step (t, B) is grounded ($B \subseteq \mathcal{B}$). This allows the search to escape local optima by making dramatic changes to plans, similar to mutation operators in evolutionary search for planning (Justesen et al., 2018; Perez et al., 2013).

Both successor functions are implemented using an LLM with appropriate prompts (Appendix C). The number of successors or branching factor is a crucial hyper parameter that allows us to control the allocation of the test-time compute budget – a larger branching factor allows a greater exploration of the plan space $\mathcal{P}$. Given a branching factor of f, we sample f times from the LLM (GPT-4o) to generate f successors.

3.1.2. PLAN GROUNDING

To guide the successor function and aid node scoring (for ranking), we need to ground each step intent in symbols found in the codebase that might be used to implement each step. This is achieved through the grounding function $g : \mathcal{L} \to \mathbb{P}(\mathcal{B})$ which maps a natural language intent t to relevant symbols in the codebase $B \subseteq \mathcal{B}$. This is challenging due to the semantic gap between high-level natural language intents and low-level code implementations (Liang et al., 2022). Rather than attempting direct intent-to-code matching, we bridge this gap through an intermediate representation.

We use a retrieval-based approach for implementing the g that reduces the challenging intent-to-code grounding problem to an easier intent-to-intent matching problem. For each symbol $b \in \mathcal{B}$, we use a lightweight language model to generate synthetic intents that describe potential uses of the symbol (e.g., "symbol b can be used to..."). These synthetic intents are then embedded into a vector space using an embedding model $e : \mathcal{L} \to \mathbb{R}^d$. Given a plan step intent t, the grounding function retrieves the synthetic intents with the highest cosine similarity and returns the corresponding symbols. We return top-5 symbols after de-duplicating matches to the same symbol via alternate synthetic intents. We use GPT-4o-mini to generate the synthetic intents and text-embedding-3-large to compute intent embeddings. (Examples in Table 4).

Having grounded symbols as part of the plan allows the successor function to identify infeasible or unrealizable steps in the plan and modify them. Similarly, the scoring function uses grounded symbols to score realizable plans more favorably than plans with unrealizable steps.

3.2. Scoring Function and Exploration Strategies

The scoring function $h : \mathcal{P} \to \mathbb{R}$ plays two crucial roles in our approach. First, it enables informed search algorithms (e.g. best-first search) by guiding exploration toward promising regions of the plan space. Second, it allows selecting the most promising plans to pass to downstream code generation, even when using uninformed search strategies like depth-first search which simply rely on node expansion order via a stack data structure to determine the next plan to mutate.

Designing an effective ranking function is challenging because we need to impose an ordering over the plan space that correlates with two key properties: (1) the likelihood that a plan achieves the user's intent, and (2) the feasibility of implementing each step with the grounded symbols. Unlike traditional planning scenarios where the scoring function emerges naturally from the environment, we must construct a scoring function that can evaluate plans without executing them.

3.2.1. Scoring Function Variants

Symbol Diversity Scorer implements a heuristic based on symbol coverage. For a plan $p = [(t_1, B_1), ..., (t_n, B_n)]$:

$$h_{\text{sym}}(p) = |\bigcup_{i=1}^{n} B_i| \tag{1}$$

This rewards plans that incorporate a diverse set of symbols from the codebase, based on the intuition that effective plans likely require integrating multiple components. While simple, this approach provides a baseline that encourages thorough exploration of available functionality.

Decomposed Likert Scorer draws inspiration from recent work showing that decomposing evaluation into fine-grained criteria improves assessment reliability (Saad-Falcon et al., 2024). We construct a scoring function that evaluates both plan-level and step-level properties using a large language model as a judge. Given the user query, the plan, and symbol definitions, we ask an LLM to produce the following judgement scores one a 7-point Likert scale (Likert, 1932):

- A plan-level accuracy score l_p assessing whether the plan solves the user request
- Step-level feasibility scores $l_1, ..., l_n$ evaluating whether each step intent t_i is realizable with the grounded symbols B_i

During informed search, we aggregate these scores into a single value to pick the next node for exploration:

$$h_{\text{likert}}(p) = \frac{1}{2} \cdot \left(l_p + \frac{1}{n} \sum_{i=1}^{n} l_i \right) \tag{2}$$

However, for final plan selection after search completes, we empirically found that a hierarchical sorting approach is more effective. Plans are first sorted by their plan-level score l_p, with ties broken by the average step-level score $\frac{1}{n}\sum_{i=1}^{n} l_i$. This two-level sorting ensures we prioritize plans that are likely to achieve the user's intent while using step-level feasibility as a secondary criterion.

Oracle Scorer. For some of our ablations, we use symbol recall (% of ground truth symbols in generated plans) to score plans. Since this requires reference code this is not a practical setting but allows us to study the effect of components like successor function and tree-search algorithms on plan search performance in a controlled setting.

3.2.2. Exploration Strategies

Our framework allows for plugging in any tree-search algorithm to guide the exploration of plan space $\mathcal{P}$. We primarily use best-first search in our experiments to make use of the scoring function for informed exploration while using depth-first search as an uninformed search baseline. We do not use breadth-first search because it is particularly wasteful for monotonic successor function as it spends most of its search budget on early stage incomplete plans. We leave more complex algorithms like MCTS for future work.

4. Experiments

Benchmark. We evaluate our plans and code generated from our plans using the LongCodeArena (LCA) bench-

Table 2. **Comparing our plan based code generation to alternative approaches.** Approaches are sorted by average amount of context usage. Using a fraction of the context, Plan Search is competitive with adding the entire codebase into the LLM context and significantly outperforms ReAct based planning.

		Overlap Score	
Context Fill	Avg. Tokens	Best-of-5	Average
Instruction Only	250	42.3	32.2
ReAct	4,831	47.2 (+5.0)	39.8 (+7.6)
Plan Search (ours)	5,473	53.9 (+11.7)	48.0 (+15.8)
Full Repo	121,262	58.7 (+16.5)	49.9 (+17.6)

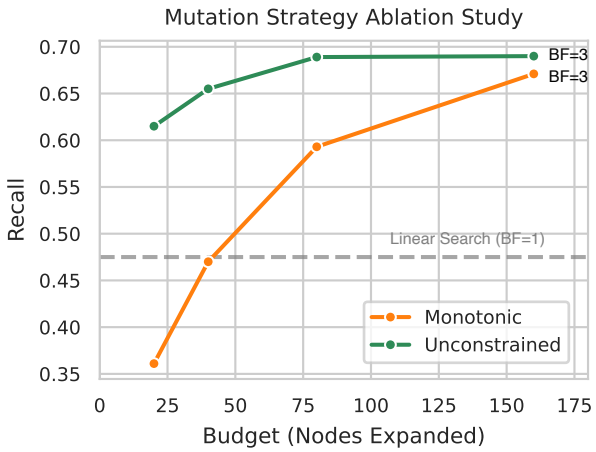

Figure 5. **Unconstrained mutation outperforms monotonic mutation, especially at lower budgets**. Here, we show the symbol recall (% of ground truth symbols in the generated plans) of each mutation strategy using best-first search with the oracle scoring function and branching factor of 3. This figure also illustrates gains from scaling test-time compute (by increasing budget).

mark[1]. Unlike traditional code generation benchmarks that focus on self-contained competition style programming problems that only require knowledge of a programming language and its primitives, LongCodeArena tasks require understanding and using an external codebase to solve the user query. Each task provides a user query, the codebase required to implement the solution, and a reference solution making it more suitable for library-style code-use evaluations compared to the popular SWE-Bench which focuses on editing the codebase itself instead of using it as a library.

Metrics. To evaluate the quality of generated code, we use the API Overlap metric introduced in LongCodeArena. This metric measures the recall of library symbols in the reference solution within the generated code. Specifically, for both the reference and generated code, we extract all symbols from their abstract syntax trees using the `tree-sitter` library and filter for symbols that belong to the target repository. The overlap score is then computed as the percentage of reference symbols that appear in the generated code. This metric captures how well the generated code utilizes the appropriate functionality from the codebase, while being robust to superficial differences in implementation.

For all experiments involving code generation, we sample 5 solutions per configuration and report both the best-of-5 and average overlap scores. The best-of-5 score reflects the best performance achieved by the downstream LLM that translates our plans to code in 5 independent tries, while the average score indicates average across those tries or expected performance. When evaluating plans directly (without code generation), we measure plan recall – the percentage of symbols from the reference solution that appear in the retrieved symbols of any plan step.

Evaluation Overview. We systematically evaluate our system components: (Section 4.1) compares the effectiveness of generating code conditioned on our grounded plans with alternatives, (Section 4.2) the impact of different successor functions, (Section 4.3) the choice of search strategy, and (Table 3) the effectiveness of different ranking functions. We conclude by demonstrating that our searched plans can help weaker language models match the performance of stronger models on these tasks in Section 4.5 and enable progress on hard tasks where even a frontier model (GPT-4o) with a context window full of repository context makes little progress (Section 4.6). Codegen prompts are in Appendix E.

[1]LCA contains multiple tracks. We use "Library-based Code Generation" track.

4.1. System-Level Comparisons

First, we evaluate plan search as part of the end-to-end system that generates code given a user query. This compares the overlap scores for the code generated from our plans to alternative approaches. We use GPT-4o (128K context window) for both plan search and for generating code from plans. Specifically, we compare the following approaches:

- **Instruction Only:** The model receives only the user query with no additional context from the codebase.
- **ReAct:** In our plan search framework, a ReAct baseline is equivalent to setting branching factor to 1 with a monotonic successor function resulting in a linear chain instead of a tree of plans. The final plan is provided as context for code generation.
- **Plan Search:** Our approach using unconstrained successor function, informed best-first search (branching factor=3, budget=80), and the symbol diversity scorer. The resulting plan is provided as context for code gen-

eration. For both ReAct and PlanSearch we use a maximum tree depth (chain length for ReAct) of 20.

- **Full Repo:** The entire repository is provided as context for code generation to establish an upper bound that fully utilizes the LLM's context window.

As shown in Table 2, the instruction-only baseline achieves an overlap score of 42.3% (best) and 32.2% (average), demonstrating that models have some ability to guess appropriate API usage from instructions alone. ReAct (linear search) improves upon this baseline (+5.0% best, +7.6% average) by actively searching the codebase. Tree-structured plan search outperforms both baselines (+11.7% best, +15.8% average over instruction-only) while using only 4.3% of the context window. Notably, this performance approaches that of using the full repository as context, despite using less than 5% of the available context budget. This demonstrates that our search can construct highly effective plans that capture precisely parts of the codebase needed to solve each task.

4.2. Successor Function Ablation

We evaluate how the choice of successor function impacts plan search performance. We compare two variants of the successor function: (i) *monotonic*, which can only add new steps and (ii) *unconstrained*, which can modify any part of the plan (see Section 3.1). To isolate the effect of the successor function, we fix other components of the system: informed search (best-first) with branching factor of 3, maximum tree depth of 20, and use an oracle ranker that scores plans based on their recall of ground truth symbols for the user query. We use GPT-4o to guide the plan search and vary the search budget (nodes expanded) from 20 to 160.

Figure 5 shows plan's symbol recall (percentage of ground truth symbols found) as a function of search budget. First, note that for both successor functions, performance improves with increasing search budget. This re-affirms the role of scaling test-time compute to improving reasoning performance. Next, we see that that unconstrained mutation consistently outperforms monotonic mutation across all budgets, with the gap being particularly pronounced at lower budgets (+30% at budget=20). The unconstrained successor achieves its peak performance with fewer steps as compared to monotonic successor suggesting more efficient exploration of the plan space through non-monotonic changes. We also show the performance of ReAct (linear search with a monotonic successor as described in Section 4.1) for reference. Tree search significantly outperforms linear search, regardless of the choice of successor function.

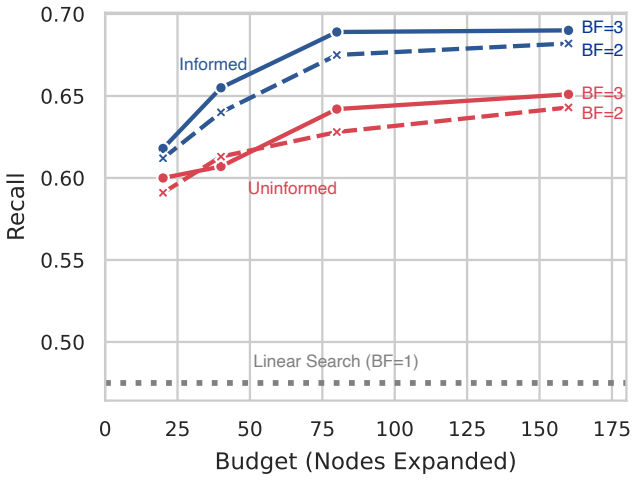

Figure 6. **Comparison of Search Strategies.** Informed (best-first) search outperforms uninformed (depth-first) and linear search strategies and performance improves with branching factor (BF), especially for informed search.

Table 3. **Comparing scoring functions**. While diversity scorer scores slightly better on the overlap metrics, an LLM judge prefers the plans chosen by the Likert scorer suggesting higher fidelity of Likert-chosen plans to the original user query.

		Overlap Score	
Scoring Fn.	Win Rate	Best-of-N	Average
Diversity	35%	49.8	41.9
Likert	65%	47.2	39.8

4.3. Traversal Ablation

We now investigate how different search strategies and branching factors affect plan quality. We compare informed search (best-first) against uninformed search (depth-first) as well as linear search. For this experiment, we use the unconstrained successor function and budget of 160 with a maximum tree depth of 20. As in the previous experiment, we use an oracle ranker for informed search to establish an upper bound on achievable performance.

As shown in Figure 6, informed search significantly outperforms uninformed and linear search strategies with branching factor of 3 showing healthy gains over 2. This is because informed search makes a more efficient use of the compute budget by exploring more promising parts of the plan space $\mathcal{P}$. Both tree-search strategies do much better than linear search which does not explore alternative solutions.

4.4. Scoring Function Ablation

Previous ablations used an oracle scorer to establish the potential of different search strategies. In practice we need

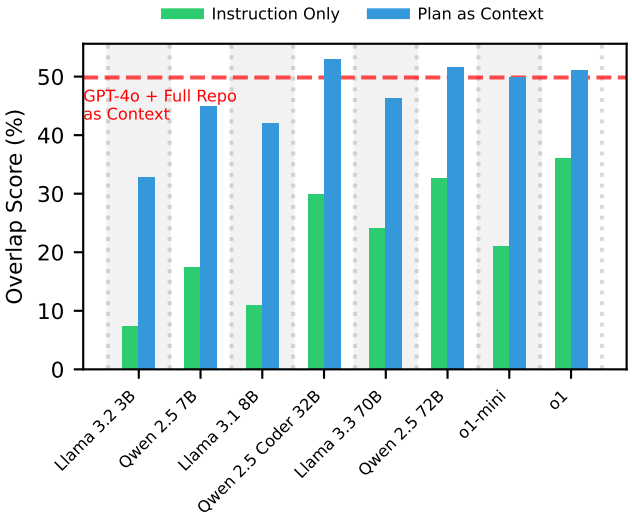

Figure 7. **Our plans consistently improve performance across all models.** Qwen 2.5 Coder 32B with our plans exceeds GPT-4o's full-repo performance despite conditioning on 120k fewer context tokens. Even models stronger than GPT-4o (e.g., O1) benefit from our GPT-4o-generated plans. The red line shows GPT-4o performance when given the full repository as context.

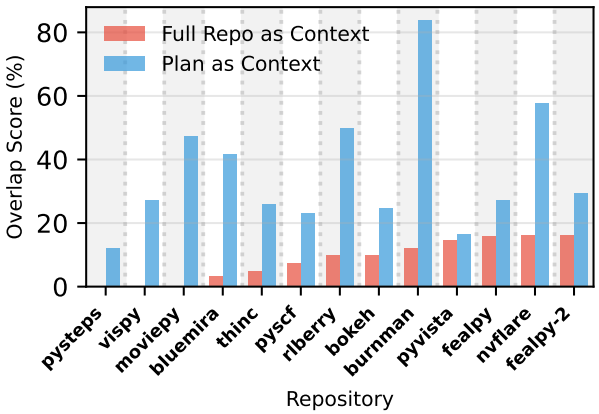

Figure 8. **Our plans enable progress on hard tasks where even full-repo context performed poorly.** Conditioning on tree-searched plans shows gains on the hardest 10% of tasks where GPT-4o with full-repo context performed poorly.

to score plans without access to ground truth symbols. We compare our symbol diversity scorer and decomposed likert scorer from Section 3.2.1. For this evaluation, we first generate candidate plans using uninformed depth-first search with a budget of 160 and the monotonic successor function and then use each scoring function to select the best plan among these candidates. The selected plan for each scoring function is then given to GPT-4o to generate the code.

We evaluate the scoring functions in two ways. First, we do a pairwise comparison between code generated by using the two scoring functions using an LLM judge. Given the code generated from both scoring functions, we ask an LLM judge (GPT-4o) to pick the code that better matches the reference code. We then compute the win rate of each scoring function. To ensure reliable evaluation, we collect 6 judgments per pair and take a majority vote. Second, we compute overlap scores using reference code. Table 3 shows that while the diversity scorer achieves slightly higher overlap scores (49.8% best-of-N vs 47.2%), the plans selected using the Likert scorer are preferred by LLM judge in pairwise comparisons (65% win rate). We hypothesize that this might be due to the Likert scorer's ability to pick plans with more accurate decomposition of the user query into step level intents than the diversity scorer which does not consider the fidelity of the plans to the original user query.

4.5. Enhancing Other LLMs with Searched Plans

Next, we examine whether plans searched by our system can enhance the performance of other language models, particularly smaller open-source models. We test a range of open models models from 3B to 70B parameters, including both general-purpose LLMs (Llama, Qwen) and models specifically fine-tuned for code (Qwen Coder), as well as OpenAI's reasoning models (O1, O1-mini). For each model, we compare performance with instruction only (i.e. no repo context) and performance with instructions augmented by the plans generated by our approach as repo context.

Figure 7 shows several striking results. First, our searched plans consistently and significantly improve performance across all models tested. The magnitude of improvement is particularly dramatic for smaller models - Llama 3.2 3B improves from 7.4% to 32.9% overlap score when given our plans, while Qwen 2.5 7B improves from 17.5% to 45.0%. Qwen 2.5 Coder 32B with our plans (53.0% overlap) outperforms GPT-4o with full repository access (49.9% overlap), despite conditioning on 95% less context. This suggests that our plans are providing a more efficient form of context than raw repository content, enabling smaller models to match or exceed the performance of much larger ones.

Our plans improve performance even for models that are stronger than the one used to generate the plans. For instance, O1, which achieves a 36.1% score with instructions alone (significantly better than GPT-4o's instruction-only performance), sees substantial gains from our GPT-4o-generated plans, improving to 51.1%, while O1-mini improves from 21.0% to 50.0% overlap given our plans.

4.6. Impact of Plans on Hard LongCodeArena Tasks

We analyze performance on the most challenging tasks in LongCodeArena (the 10% of tasks for which GPT-4o with the entire repository as context makes the least progress). Figure 8 compares providing the full repository as context (red) against using our tree-searched plans as context (blue).

For each approach, we sample 5 programs conditioned on the context per task and show the average performance using the overlap metric. On these challenging tasks, GPT-4o with full repository context (128K tokens) struggles significantly, achieving average overlap scores below 20% across all repositories. In contrast, our tree-searched plans (unconstrained successor, branching factor=3) enable substantially better performance across all tasks. For instance, on the `burnman` task, our approach achieves an average overlap of 83.75%, while full repository context manages only 12%.

5. Conclusion

MUTAGREP automatically enriches user queries with repo-grounded plans found through execution-free tree search. Our system decomposes high-level requests into detailed plans where each step pairs natural language intent with relevant codebase symbols. Through experiments on LongCodeArena, we demonstrated that our plans: (1) are effective as context for code generation; (2) enable weaker models to match stronger models' performance; and (3) enable progress on challenging tasks where even frontier models with full repository context struggle. Our results show that grounded plan search is a promising direction for improving code-use while maintaining efficiency and interpretability.

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here.

References

Bogomolov, E., Eliseeva, A., Galimzyanov, T., Glukhov, E., Shapkin, A., Tigina, M., Golubev, Y., Kovrigin, A., van Deursen, A., Izadi, M., and Bryksin, T. Long code arena: a set of benchmarks for long-context code models, 2024. URL https://arxiv.org/abs/2406.11612.

Bonet, B. and Geffner, H. Planning as heuristic search. *Artificial Intelligence*, 129(1–2):5–33, 2001.

Brown, B., Juravsky, J., Ehrlich, R., Clark, R., Le, Q. V., Ré, C., and Mirhoseini, A. Large language monkeys: Scaling inference compute with repeated sampling, 2024. URL https://arxiv.org/abs/2407.21787.

Ehrlich, R., Brown, B., Juravsky, J., Clark, R., Ré, C., and Mirhoseini, A. Codemonkeys: Scaling test-time compute for software engineering, 2025. URL https://arxiv.org/abs/2501.14723.

Gupta, T., Weihs, L., and Kembhavi, A. Codenav: Beyond tool-use to using real-world codebases with llm agents, 2024. URL https://arxiv.org/abs/2406.12276.

Hoffmann, J. and Nebel, B. The ff planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. R. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Justesen, N., Mahlmann, T., Risi, S., and Togelius, J. Playing multiaction adversarial games: Online evolutionary planning versus tree search. *IEEE Transactions on Games*, 10(3):281–291, 2018. doi: 10.1109/TCIAIG.2017.2738156.

Kuratov, Y., Bulatov, A., Anokhin, P., Rodkin, I., Sorokin, D., Sorokin, A., and Burtsev, M. Babilong: Testing the limits of llms with long context reasoning-in-a-haystack. In *Advances in Neural Information Processing Systems*, volume 37, 2024.

Li, J., Le, H., Zhou, Y., Xiong, C., Savarese, S., and Sahoo, D. Codetree: Agent-guided tree search for code generation with large language models. *arXiv preprint arXiv:2411.04329*, 2024.

Li, M., Zhao, Y., Yu, B., Song, F., Li, H., Yu, H., Li, Z., Huang, F., and Li, Y. Api-bank: A comprehensive benchmark for tool-augmented llms. *arXiv preprint arXiv:2304.08244*, 2023.

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Lago, A. D., Hubert, T., Choy, P., de Masson d'Autume, C., Babuschkin, I., Chen, X., Huang, P.-S., Welbl, J., Gowal, S., Cherepanov, A., Molloy, J., Mankowitz, D. J., Robson, E. S., Kohli, P., de Freitas, N., Kavukcuoglu, K., and Vinyals, O. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022. doi: 10.1126/science.abq1158. URL https://www.science.org/doi/abs/10.1126/science.abq1158.

Liang, W., Zhang, Y., Kwon, Y., Yeung, S., and Zou, J. Mind the gap: Understanding the modality gap in multimodal contrastive representation learning. In *NeurIPS*, 2022. URL https://openreview.net/forum?id=S7Evzt9uit3.

Likert, R. A technique for the measurement of attitudes. *Archives of Psychology*, 22(140):1–55, 1932.

Ma, Z., Huang, W., Zhang, J., Gupta, T., and Krishna, R. m&m's: A benchmark to evaluate tool-use for multi-step multi-modal tasks. In *European Conference on Computer Vision (ECCV)*, 2024.

McDermott, D. V. Using regression-match graphs to control search in planning. *Artificial Intelligence*, 109(1–2):111–159, 1999.

Perez, D., Samothrakis, S., Lucas, S., and Rohlfshagen, P. Rolling horizon evolution versus tree search for navigation in single-player real-time games. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, GECCO '13, pp. 351–358, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450319638. doi: 10.1145/2463372.2463413. URL https://doi.org/10.1145/2463372.2463413.

Saad-Falcon, J., Vivek, R., Berrios, W., Naik, N. S., Franklin, M., Vidgen, B., Singh, A., Kiela, D., and Mehri, S. Lmunit: Fine-grained evaluation with natural language unit tests, 2024. URL https://arxiv.org/abs/2412.13091.

Trinh, T. H., Wu, Y., Le, Q. V., He, H., and Luong, T. Solving olympiad geometry without human demonstrations. *Nat.*, 625(7995):476–482, 2024. doi: 10.1038/S41586-023-06747-5. URL https://doi.org/10.1038/s41586-023-06747-5.

Wang, E., Cassano, F., Wu, C., Bai, Y., Song, W., Nath, V., Han, Z., Hendryx, S., Yue, S., and Zhang, H. Planning in natural language improves llm search for code generation. *arXiv preprint arXiv:2409.03733*, 2024a.

Wang, X., Chen, Y., Yuan, L., Zhang, Y., Li, Y., Peng, H., and Ji, H. Executable code actions elicit better llm agents. *arXiv preprint arXiv:2402.01030*, 2024b.

Xia, C. S., Deng, Y., Dunn, S., and Zhang, L. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint*, 2024.

Yang, J., Jimenez, C. E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K. R., and Press, O. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://arxiv.org/abs/2405.15793.

This appendix provides additional details, analyses and experimental results to supplement the main paper. Appendix A presents qualiative examples of the synthetic intents used to ground plans in a codebase. Appendix B presents an in-depth analysis of our system's performance on challenging tasks from LongCodeArena. Appendix C shows the complete prompts used for our successor functions. Appendix D details the prompts used for our scoring functions. Appendix E provides the prompts used for code generation across different experimental settings.

A. Qualitative Examples of Synthetic Intents

Table 4: **Qualitative Examples of Synthetic Intents**: We show randomly selected examples of the top 3 closest synthetic intents by embedding distance (using OpenAI's `text-embedding-3-large`) to a query intent. Each synthetic intent is generated conditioned on a symbol from the codebase (using `GPT-4o-mini`). To ground plan steps during plan search, the intent from the plan step is matched to synthetic intents and therefore to the symbols corresponding to the synthetic intents.

Repository	Intent	Top-3 Closest Synthetic Intents (`Symbol`)
pybamm	Add a no SEI submodel.	*I need to create a new discretisation instance without providing a mesh.* (`Discretisation`) *I want to create an isothermal thermal submodel for my simulation.* (`Isothermal`) *I want to assign parameter values to a specific model.* (`process_model`)
	Add a constant porosity submodel.	*I want to create an instance of the constant concentration diffusion model.* (`ConstantConcentration`) *I want to create an isothermal thermal submodel for my simulation.* (`Isothermal`) *I want to initialize a constant concentration model for the diffusion process with specific parameters.* (`ConstantConcentration`)
	Add an isothermal thermal submodel.	*I want to create an isothermal thermal submodel for my simulation.* (`Isothermal`) *I need to set up the Isothermal model for my thermal simulations.* (`Isothermal`) *I need to gather all temperature variables associated with the isothermal submodel.* (`get_fundamental_variables`)
dd4hep	Set up a particle gun with specified parameters.	*I want to set up a particle gun in the simulation to start generating particles.* (`setupGun`) *I need to configure the particle gun with specific parameters such as name, particle type, and energy level.* (`setupGun`) *I want to customize the position and multiplicity settings for the particle gun in the simulation.* (`setupGun`)

Continued on next page

Continued from previous page

Repository	Intent	Top-3 Closest Synthetic Intents (`Symbol`)
	Set up a tracker for the simulation.	*I want to set up a tracking field for my particle simulation using a specific configuration.* (`setupTrackingFieldMT`) *I want to set up the construction of the detector in the simulation.* (`detectorConstruction`) *I want to configure the tracking field setup for my Geant4 simulation.* (`setupTrackingField`)
	Set up event actions for particle printing.	*I am looking to set up a generator action for particle generation in my application.* (`GeneratorAction`) *I want to set up a particle gun in the simulation to start generating particles.* (`setupGun`) *I would like to use the tracking action functionality to monitor particle tracks in my experiment.* (`TrackingAction`)
fealpy	Create a uniform time mesh for the simulation.	*I want to generate the initial mesh for my 2D time harmonic solver.* (`init_mesh`) *I want to ensure that the mesh is refined uniformly to improve simulation accuracy.* (`init_mesh`) *I want to create a uniform triangular mesh to use in my analysis.* (`init_mesh`)
	Solve the linear system to update the solution at the current time step.	*I need to update my solution by solving the linear system after applying Dirichlet boundary conditions.* (`solve`) *I need to iterate through time steps and update my model's solutions.* (`time_integration`) *I need to update the state of my model variables after solving the system.* (`solve`)
	Advance to the next time level in the time mesh.	*I want to progress the time in my algorithm by moving to the next time level.* (`next_time_level`) *I want to advance to the next time level in the simulation.* (`next_time_level`) *I want to progress to the subsequent time level in the timeline.* (`next_time_level`)
nplab	Create an experiment class that involves a shutter and a spectrometer.	*I want to initialize a new shutter instance in my experiment setup.* (`Shutter`) *I want to prepare a shutter for my nanophotonics experiments.* (`Shutter`) *I need to construct a shutter object to manage exposure times.* (`Shutter`)

Continued on next page

Continued from previous page

Repository	Intent	Top-3 Closest Synthetic Intents (`Symbol`)
	Initialize and display the GUI application.	*I want to initialize a new GUI widget that will display a plot.* (`Widget`) *I want to initialize the user interface for the spectrometers in the application.* (`_init_ui`) *I need to initialize a GUI component that displays spectrometer controls.* (`SpectrometersUI`)
	Define properties for irradiation time and wait time.	*I need to configure the integration time and delay settings for my spectrometer to ensure accurate time series measurements.* (`update_time_series_params`) *I need to expose the instrument for a set amount of time and ensure it blocks until the exposure completes.* (`expose`) *I want to specify the duration for which the spectrometer should collect data during a measurement.* (`set_integration_time`)
python-sc2	Manage drones to gather minerals if vespene gas is above a certain threshold or Zergling speed upgrade is pending.	*I want to check if my unit is currently gathering resources from a mineral field or vespene geyser.* (`is_gathering`) *I need to identify the units that are engaged in gathering minerals or vespene.* (`gathering`) *I need to direct a unit to gather either minerals or gas for my economy.* (`gather`)
	Research Zergling speed upgrade if conditions are met.	*I need to queue an upgrade research for my unit.* (`research`) *I need to determine how fast my unit can move considering the effects of any active upgrades.* (`calculate_speed`) *I want to start researching an upgrade if the necessary tech building is ready.* (`research`)
	Draw a creep pixelmap for debugging purposes.	*I want to check if there is creep on a specific grid point in the game.* (`has_creep`) *I want to output debug information by drawing a box around a game unit.* (`debug_box2_out`) *I want to draw a visual line between two points in my game for debugging purposes.* (`debug_line_out`)

Continued on next page

Continued from previous page

Repository	Intent	Top-3 Closest Synthetic Intents (`Symbol`)
basilisk	Initialize and execute the simulation within the scenario execution function.	*I need to initialize the simulation and prepare all modules for execution.* (`SimBaseClass`) *I want to execute a simulation by assigning the appropriate execution function.* (`setExecutionFunction`) *I need to prepare my simulation for execution by initializing all required data structures and parameters.* (`SimBaseClass`)
	Import necessary modules and set up file paths for the simulation.	*I need to initialize the simulation and prepare all modules for execution.* (`SimBaseClass`) *I want to configure my simulation environment with the correct paths and logger setup on initialization.* (`SimBaseClass`) *I need to ensure that all modules in the simulation are properly self-initialized.* (`InitializeSimulation`)
	Define a function to execute the simulation scenario, including configuring stop time and initializing the simulation.	*I want to define an execution function that will run my simulation instance.* (`setExecutionFunction`) *I want to define the parameters for running a simulation, including the creation and execution functions.* (`SimulationParameters`) *I want to define how long my simulation should run by setting the stop time.* (`ConfigureStopTime`)

B. More Analysis of Hard Tasks on LongCodeArena

To better understand the robustness of our approach, we analyze worst-case and average-case performance on the most challenging tasks in LongCodeArena (bottom 10th percentile by full-repository performance). Figure 9 compares three approaches using GPT-4o as the code generator: providing the full repository as context (red), using ReACT-generated plans (orange), and using our tree-searched plans (blue). For each approach, we sample 5 solutions per task and show both the average performance (top of bars) and minimum performance (bottom of error bars) using the API overlap metric.

Our approach consistently outperforms ReACT-style planning, showing better average performance on 11 out of 13 tasks. More importantly, the worst-case performance with our plans (indicated by the bottom of the blue bars) often exceeds the average performance of both baselines, suggesting that tree-searched plans lead to more reliable code generation. This is particularly evident in repositories like moviepy, where our approach's minimum performance (45.45%) far exceeds both the ReACT average (12.73%) and full repository average (0%).

These results demonstrate that systematic tree search produces more robust plans than either naive context inclusion or linear planning approaches, particularly on challenging tasks where standard approaches struggle to make progress.

C. Successor Function Prompts

Our successor functions rely on prompts to guide the LLM in mutating plans. Figure 10 shows the prompt template used for the monotonic successor function, which can only add new steps while preserving existing ones. Figure 11 shows the prompt for the unconstrained successor function, which can modify any part of the plan.

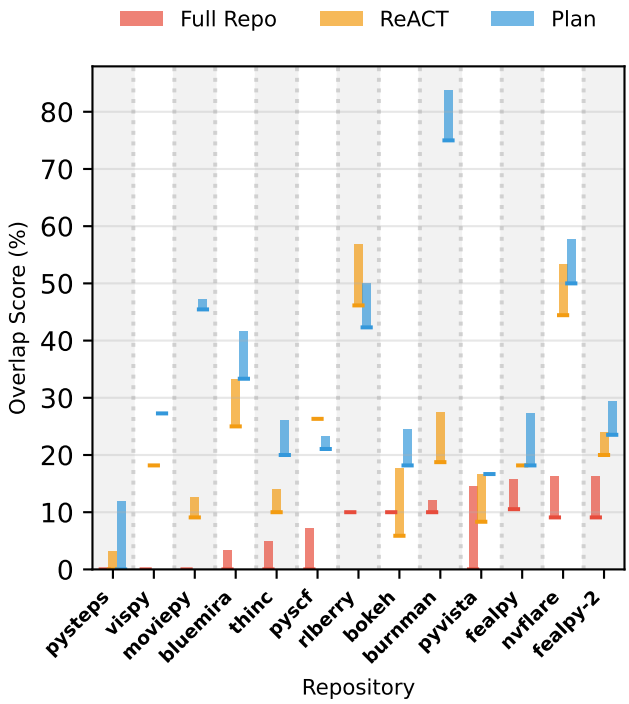

Figure 9. Performance comparison on the most challenging LongCodeArena tasks (bottom 10th percentile by full-repo performance). For each repository, we show the performance of GPT-4o when given: the full repository as context (red), ReACT-generated plans (orange), or our tree-searched plans (blue). Bars show average performance across 5 samples, while the bottom of error bars indicates worst-case performance. Our tree-searched plans (using unconstrained successor function and branching factor=3) consistently outperform both baselines, with worst-case performance often exceeding the baselines' average performance. All scores are API overlap percentages measuring alignment with reference solutions.

```
PROMPT_TEMPLATE = jinja2.Template(
    """# Example 1
<user_request>
    "I'm planning to revamp my home interior, using '417814-input.png' as reference. I want to see how the space would appear if the carpet was replaced with a wooden
floor. Also, could you determine the total number of objects present in the image once the modifications are made?"
</user_request>


<plan>
</plan>


<proposed_edit>
    <step number="0">
        <description>Modify the image to replace the carpet with a wooden floor.</description>
    </step>
</proposed_edit>


# Example 2
<user_request>
    "I'm planning to revamp my home interior, using '417814-input.png' as reference. I want to see how the space would appear if the carpet was replaced with a wooden
floor. Also, could you determine the total number of objects present in the image once the modifications are made?"
</user_request>


<plan>
    <step number="0">
        <description>Modify the image to replace the carpet with a wooden floor.</description>
        <feedback>
            <symbol_name>mnm.tool_api.image_editing</symbol_name>
            <satisfiable>True</satisfiable>
            <justification>
                The function `image_editing` is specifically designed to modify images based on a given prompt, and it will replace the carpet with a wooden floor as
requested.
            </justification>
        </feedback>
    </step>
</plan>


<proposed_edit>
    <step number="1">
        <description>Detect objects in the modified image.</description>
    </step>
</proposed_edit>




# Instructions
You are given an incomplete plan and your task is to propose a modification to the plan to get it closer to satisfying the user request.
Follow the format shown in "Example 1" and "Example 2" above.


Ensure the step number is a single integer like 0, 1, 12, 42, etc. Do not use letters like 5a or decimals like 0.5.
Produce valid XML and do not include any other text or comments that would break XML parsing.




<user_request>
{{ user_request }}
</user_request>


<plan>
{% for step in plan.steps %}
<step number="{{ step.index }}">
    <description>{{ step.content }}</description>
    <feedback>
        <symbol_name>{{ step.search_result.symbol_name }}</symbol_name>
        <satisfiable>{{ step.search_result.satisfies_intention }}</satisfiable>
        <justification>
            {{ step.search_result.justification }}
        </justification>
    </feedback>
</step>
{% endfor %}
</plan>""",
    undefined=jinja2.StrictUndefined,
)
```

Figure 10. **Monotonic Successor Function Prompt**

```
UNCONSTRAINED_PROMPT_TEMPLATE = jinja2.Template(
    """# Instructions
You are given a plan for implementing a user request. Your task is to propose a modified version of the plan that might better satisfy the user request.


For each step in the plan, you will be given feedback from a search tool.
The feedback consists of:
- Symbols in the codebase that are most likely to help accomplish the step
- The signatures of those symbols
Use the feedback to help you make modifications to the plan.
For example, you may be able to intuit from reading the signatures that none of the symbols are relevant to the step, and thus remove the step and replace it with a new step.
Or, you may see that it seems like the step is not necessary, and thus remove it.
Or, you may see that the step seems feasible to accomplish and requires no changes.




You can make any combination of the following modifications:
- Add new steps anywhere in the plan (e.g. to fill in missing steps)
- Remove existing steps (e.g. to remove unnecessary steps)
- Modify existing steps to be more specific or accurate (e.g. based on the feedback from the search tool)
- Reorder steps to improve the execution flow (e.g. based on the feedback from the search tool)


Format your output as an XML document with the following structure:
```xml
<thought>
<-- YOUR THOUGHT PROCESS for modifying the plan -->
</thought>
<plan>
    <step number="0">
        <description>Modify the image to replace the carpet with a wooden floor.</description>
    </step>
    <step number="1">
        <description>Detect objects in the modified image.</description>
    </step>
    <step number="stepnum">
        <description>stepdesc</description>
    </step>
</plan>
```
Here, `stepnum` is an integer and `stepdesc` is a string.
IMPORTANT: The step number must be an integer enclosed in double quotes.
IMPORTANT: Your output must be valid XML and must not contain any other text or comments that would break XML parsing.

Your Task
The user request is: "{{ user_request }}"

Here is a sample of some of the symbols in the codebase:
{% for symbol in starting_symbols %}
- {{ symbol.name }}: {{ symbol.filepath }}
{% endfor %}

Here is the repository tree:
{{ repo_tree }}

Remember that the list above is just a starting point to give you an idea of what functionality is available in the codebase.
There are many more symbols in the codebase that are not listed above.

{% if plan.steps | length > 0 %}
The current plan is:
<plan>
{% for step in plan.steps %}
<step number="{{ step.index }}">
    <description>{{ step.content }}</description>
    <feedback>
        {% for signature in get_signatures(step.search_result) %}
        - {{ signature }}
        {% endfor %}
    </feedback>
</step>
{% endfor %}
</plan>
Propose a modified plan that better accomplishes the user request.
{% else %}
Currently, the plan is empty. Propose an initial plan (it can be incomplete), that we can work on modifying.
{% endif %}

Remember the following guidelines:
- The step number must be an integer enclosed in double quotes.
- Your output must be valid XML and must not contain any other text or comments that would break XML parsing.
""",
    undefined=jinja2.StrictUndefined,
)
```

Figure 11. **Unconstrained Successor Function Prompt**

```
judge_prompt_template = jinja2.Template(
    """# Instructions
You will be given a user request and a plan to accomplish that user request with a codebase.
Each step of the plan will have a list of symbols that are relevant to that step.

Judge the entire plan based on the following criteria on a scale of 1 to 7:
- This plan solves the user request; it is not missing any necessary steps.

Judge each step based on the following criteria on a scale of 1 to 7:
- This step is achievable with the symbols found; you could write code to implement this step using the listed symbols.

The scale indicates your degree of agreement with the statement.
- 1: Strongly disagree
- 2: Disagree
- 3: Slightly disagree
- 4: Neutral
- 5: Slightly agree
- 6: Agree
- 7: Strongly agree

Your response must be in the following XML format:
<judgement>
  <plan_level>
    <solves_user_request>NUMBER</solves_user_request>
  </plan_level>
  <steps>
    <step>
      <step_index>INDEX_OF_STEP</step_index>
      <achievable_with_symbols>NUMBER</achievable_with_symbols>
    </step>
    <!-- Repeat for each step -->
  </steps>
</judgement>

ONLY output the XML.
DO NOT wrap the XML in triple backticks.
DO NOT provide any other output.

# User Request
{{ plan.user_query }}

# Code Definitions
```python
{% for symbol in all_symbols_used %}
# Filepath: {{ symbol.filepath }}
# Import Path: {{ symbol.full_path }}
{{ truncated_code_display(symbol.code) }}
{% endfor %}
```

Plan
{% for step in plan.steps %}
Step {{ step.index }}
{{ step.content }}
Symbols Found
{% for symbol in step.search_result.instrumentation.symbols_considered %}
- {{ symbol.symbol.full_path }}
{% endfor %}
{% endfor %}
""",
    undefined=jinja2.StrictUndefined,
)
```

Figure 12. **Likert Scoring Function Prompt**

D. Scoring Function Prompts

The Likert scoring function uses the prompt shown in Figure 12 to evaluate plans. The prompt breaks down evaluation into two aspects: (1) whether the overall plan achieves the user's intent and (2) whether each step is feasible with its retrieved symbols. The scoring is done on a 7-point Likert scale.

E. Code Generation Prompts

The code generation prompts are designed to evaluate different approaches to providing repository context. Figure 13 shows how we present plans as structured context for code generation. Figure 14 demonstrates the full-repository baseline approach where the entire codebase is provided as context. Figure 15 shows the minimal instruction-only setting which provides no additional context beyond the user query.

```
PLAN_TO_CODE_TEMPLATE = jinja2.Template(
    """Your task is to write Python code that achieves a user query.
You will be provided a step-by-step plan for accomplishing the user query.
Use the plan to help you write code that accomplishes the user query.
Each step of the plan contains a list of suggested symbols to use in that step.
You will be provided the definition of each symbol mentioned in the plan.
{% if project_defined_elements is not none %}
You will also be provided a list of all symbols in the codebase.
{% endif %}
{% if repo_tree is not none %}
You will also be given a map of the codebase structure.
{% endif %}
You can use all of the above information to write code that accomplishes the user query.
{% if encourage_symbol_usage %}
It is important to stick to the plan as closely as possible and consider using the symbols provided for each step.
{% endif %}

Produce your output in the following format:
```python
# your code goes here
```
Do not include any other text in your output. Stick exactly to the required format.

{%- if repo_tree is not none -%}
Repository Tree
{{ repo_tree }}
{%- endif -%}

{%- if project_defined_elements is not none -%}
List of all symbols in the codebase
{%- for element in project_defined_elements %}
- {{ element }}
{%- endfor -%}
{%- endif -%}

Code Definitions
```python
{% for symbol in all_symbols_used %}
# Filepath: {{ symbol.filename }}
# Python Path: {{ symbol.full_path }}
{{ code_display(symbol) }}
{% endfor %}
```

User query
{{ user_query }}

Step-by-step plan
{% for step in plan -%}
Step {{ step.index }}
- {{ step.content }}
Symbols
{% for symbol in step.search_result.instrumentation.symbols_considered -%}
- {{ symbol.symbol.full_path }}
{% endfor %}
{% endfor %}""",
    undefined=jinja2.StrictUndefined,
)
```

Figure 13. **Plan-based Code Generation Prompt**

```
prompt = jinja2.Template(
        """Write Python code to help a user complete a task using a library.
The library is {{ repo_name }}.


A description of the library is provided below:
{{ gitingest_content }}


The user's task is: "{{ task_description }}"
Output nothing but the code to complete the task.
Wrap the code in ```python and ```.
Do not include any other text in your response.
""")
```

Figure 14. **Full-repository Context Code Generation Prompt**

```
    prompt = jinja2.Template(
        """Write Python code to help a user complete a task using a library.
The library is {{ repo_name }}.

The user's task is: "{{ task_description }}"

Using your knowledge of the library, write Python code to complete the task.
Output nothing but the code.
Wrap the code in ```python and ```.
Do not include any other text in your response.""",
        undefined=jinja2.StrictUndefined)
```

Figure 15. **Instruction-only Code Generation Prompt**