ML-TOOL-BENCH: TOOL-AUGMENTED PLANNING FOR ML TASKS

Anonymous authors
Paper under double-blind review

ABSTRACT

The development of autonomous machine learning (ML) agents capable of end-to-end data science workflows represents a significant frontier in artificial intelligence. These agents must orchestrate complex sequences of data analysis, feature engineering, model selection, and hyperparameter optimization, tasks that require sophisticated planning and iteration. While recent work on building ML agents has explored using large language models (LLMs) for direct code generation, tool-augmented approaches offer greater modularity and reliability. However, existing tool-use benchmarks focus primarily on task-specific tool selection or argument extraction for tool invocation, failing to evaluate the sophisticated planning capabilities required for ML Agents. In this work, we introduce a comprehensive benchmark for evaluating tool-augmented ML agents using a curated set of 61 specialized tools and 15 tabular ML challenges from Kaggle. Our benchmark goes beyond traditional tool-use evaluation by incorporating an in-memory named object management, allowing agents to flexibly name, save, and retrieve intermediate results throughout the workflows. We demonstrate that standard ReAct-style approaches struggle to generate valid tool sequences for complex ML pipelines, and that tree search methods with LLM-based evaluation underperform due to inconsistent state scoring. To address these limitations, we propose two simple approaches: 1) using shaped deterministic rewards with structured textual feedback, and 2) decomposing the original problem into a sequence of sub-tasks, which significantly improves trajectory validity and task performance. Using GPT-4o, our approach improves over ReAct by 16.52 percentile positions, taking the median across all Kaggle challenges. We believe our work provides a foundation for developing more capable tool-augmented planning ML agents.

1 INTRODUCTION

Autonomous agents capable of solving end-to-end machine learning (ML) tasks represent a critical frontier in artificial intelligence (Grosnit et al., 2024; Toledo et al., 2025; Yang et al., 2025; Chan et al., 2025). Such agents must be capable of doing: data preprocessing, feature engineering, model training, and hyperparameter tuning, while managing intermediate results and adapting their strategies based on the evolving context. Achieving this level of autonomy requires not only sophisticated planning, but also memory management and the capacity to coordinate multiple operations coherently. Large language models (LLMs) have recently been explored as the foundation for such agents (Grosnit et al., 2024; Chan et al., 2025; Huang et al., 2024). Early work has primarily focused on direct code generation, where the agent generates python code for completing a given ML task (Grosnit et al., 2024; Chan et al., 2025; Huang et al., 2024; Toledo et al., 2025). This paradigm has shown promise on competitive benchmarks inspired by Kaggle challenges, with some approaches achieving performance comparable to a Kaggle Master (Grosnit et al., 2024; Chan et al., 2025). Several benchmarks have also been proposed to evaluate the performance of LLMs on such tasks (Chan et al., 2025; Huang et al., 2024; Qiang et al., 2025; Jing et al., 2025; Zhang et al., 2025). However, any approach that relies on direct code generation is prone to key weaknesses: generated code is brittle (Abbassi et al., 2025; Liu et al., 2025), debugging typically requires multiple iterations, and reasoning is tightly coupled with execution (Liu et al., 2025; Chen et al., 2025).

An alternative paradigm equips LLMs with external tools, yielding tool-augmented agents that need to decide which tools to invoke and in what sequence, to solve the task. Tools offer modular, reusable

building blocks for data-science workflows: from preprocessing, to training, and evaluation. This design has proven effective in broader domains, including web navigation (Zhou et al., 2024b), operating systems (Bonatti et al., 2024), and code interpretation (Huang et al., 2024), yet its potential for ML workflows remains underexplored. Crucially, tool augmentation reformulates the problem as planning in a large action space: the agent must coordinate multi-step trajectories and retrieve and reuse intermediate artifacts (or results). Because the agent is restricted to a curated toolset, tool-augmented approaches decouple high-level reasoning from low-level code execution, improving modularity, reliability, and safety.

Existing benchmarks for tool use fall short on long-horizon planning. Most benchmarks and approaches evaluate whether agents can select the right tools and valid arguments. The Berkeley Function-Calling Leaderboard (BFCL) (Patil et al., 2025) measures single, parallel, and multiple function calling, and BFCL-v3 (Patil et al., 2025) extends this to multi-turn, multi-step settings. However, even BFCL-v3 emphasizes relatively shallow plans compared to ML workflows, which might require long-term planning, iterative refinement, and reuse of intermediate artifacts. Similarly ToolBench (Xu et al., 2023) provides a suite of diverse software tools, that span both single-step and multi-step action generation, but focuses on evaluating whether the LLM can correctly select tools and tool arguments.

In this work, we introduce ML-Tool-Bench, motivated by the lack of good benchmarks to assess planning approaches with tools in ML workflows. In particular, ML-Tool-Bench provides a benchmark to evaluate the planning capabilities of LLM agents on *tabular* Kaggle ML challenges. We introduce a curated suite of 61 tools sufficient to solve such tasks and assess performance across 15 Kaggle challenges spanning regression and classification.

We evaluate multiple agents using several different planning algorithms, on our benchmark. To enable agents to create, persist, and reuse intermediate artifacts, we adopt an in-memory, named-object management scheme: tools accept references to named objects, and agents can assign names to tool outputs. We refer to this as scratchpad-augmented planning: agents store and retrieve objects by name over multi-step trajectories, enabling tools to handle arbitrarily large or structured inputs, unlike prior benchmarks that restrict arguments to simple types (e.g., strings, integers, floats). We observe that simple methods like ReAct (Yao et al., 2023b) struggle to produce performant trajectories across our Kaggle benchmark. Monte Carlo Tree Search-based methods (Kocsis & Szepesvari, 2006; Silver et al., 2016) such as LATS (Zhou et al., 2024a), which rely on LLMs as value estimators, also underperform due to inconsistent trajectory scoring. In contrast, we propose two simple approaches: 1) combining shaped, deterministic rewards with textual feedback and 2) decomposing the original problem into a sequence of sub-tasks. These approaches outperform the baselines, yielding more performant tool trajectories. These results highlight the difficulty of autonomous ML planning and point toward tool-augmented systems that rely less on subjective LLM scoring as tool sets grow in size and complexity.

1. We introduce ML-Tool-Bench, a tool-augmented benchmark for end-to-end ML planning with 61 tools and 15 Kaggle challenges.
2. We formalize *scratchpad-augmented planning* via named-object management that supports arbitrarily large artifacts and reversible branching in search.
3. We propose *MCTS-Shaped*, an MCTS approach with shaped, deterministic rewards and targeted textual feedback, which improves trajectory validity and performance over ReAct and LATS.
4. We introduce *Hierarchical MCTS*, an approach that decomposes problems into sequenced subtasks, further improving validity and robustness. For GPT-4o, Hierarchical MCTS improves over LATS by 9.93 percentile positions on the leaderboard and over ReAct by 16.52 percentile positions (median across all competitions). For GPT-4.1-mini, it improves over MCTS-Shaped by 1.89 percentile positions, while both ReAct and LATS had a median percentile position of 0.

Together, these advances establish strong baselines for tool-augmented, end-to-end ML planning and reduce reliance on subjective LLM scoring.

2 Related Work

Machine Learning Benchmarks for AI Agents: Most of the existing Data Science and ML benchmarks, provide the LLM agent access to write code that solves the task, and evaluate its performance. Chan et al. (2025) propose MLE-bench, a curated benchmark of 75 Kaggle challenges, that test real-

world ML engineering skills. They find that OpenAI's o1-preview with the AI-Driven Exploration (AIDE) scaffolding (Jiang et al., 2025) achieves at least a level of Kaggle bronze medal in 16.9% of competitions in their benchmark. AIRA-dojo (Toledo et al., 2025) improves upon Chan et al. (2025), replacing AIDE (Jiang et al., 2025) with a different choice of operator set, to generate new candidate solutions, and using Monte Carlo Tree Search (MCTS) (Kocsis & Szepesvari, 2006) instead of greedy search, increasing the success rate of achieving a Kaggle medal from 39.6% to 47.7%. Huang et al. (2024) also propose a ML benchmark, called MLAgentBench, containing a suite of 13 tasks, where the agent is allowed to perform actions like read/write files, execute code and inspect outputs. They construct a ReAct based agent (Yao et al., 2023b) (with Claude v3 Opus) and were able to build compelling ML models on MLAgentBench with 37.5% average success rate. Qiang et al. (2025) propose MLE-Dojo, an interactive gym-style workflow for LLM agents in iterative ML engineering workflows, and build upon 200+ Kaggle challenges. To evaluate Data Science Agents, Jing et al. (2025) proposed a comprehensive benchmark that includes 466 data analysis tasks and 74 data modeling tasks, sourced from Eloquence and Kaggle competitions, and showed that state of the art LLMs and agents struggle on most tasks. Zhang et al. (2025) propose DataSciBench and demonstrate that closed source models (GPT, Claude etc.) outperform open source models on all metrics in their benchmark.

Learning in Tool augmented LLMs: Solving ML challenges solely through the invocation of a fixed set of tools, in the correct sequential order, remains relatively unexplored. Approaches such as ARTIST (Singh et al., 2025), ReTooL (Feng et al., 2025), StepTool (Yu et al., 2024), ToRL (Li et al., 2025), and ToolPlanner (Wu et al., 2024) couple reasoning and tool use for LLMs, using Reinforcement Learning to learn robust strategies for tool use. Recently, methods to fine-tune LLMs on responses containing tool usage have also been proposed (Schick et al., 2023; Qin et al., 2023; Gou et al., 2023; Patil et al., 2023).

Alternately, tree search methods (Yao et al., 2023a; Hao et al., 2023; Zhou et al., 2024a; Zhuang et al., 2023) have also been used to generate valid tool use trajectories. Zhuang et al. (2023) employs A* search, Hao et al. (2023) adopts Monte Carlo Tree Search (MCTS) and uses LLM as the world model, Zhou et al. (2024a) uses MCTS with value functions obtained from an LLM and self-reflection, and Yao et al. (2023a) explores Breadth-First Search (BFS) and Depth-First Search (DFS). However, these methods either depend on heuristic cost functions or leverage LLM feedback as a value function, and they are primarily applied to problems with relatively shallow depth. LATS (Zhou et al., 2024a) and Toolchain* (Zhuang et al., 2023) are the only approaches that explore planning with tools while the others restrict themselves to reasoning or toy domains. Feng et al. (2024) propose TS-LLM, an AlphaZero-inspired tree-search framework for LLMs that integrates a learned value function to guide decoding. The trajectories generated from tree search can further be used to fine-tune and improve the LLM, and TS-LLM has been shown to scale to tree depths of up to 64. Another approach, ReST-MCTS (Zhang et al., 2024), adopts a similar strategy to TS-LLM; however, in this case the per-step rewards are inferred directly from MCTS, whereas TS-LLM infers them using TD-λ (Sutton, 1988).

Tool Benchmarks: Benchmarks for LLM tool use largely emphasize correct tool selection and argument specification rather than extended planning. ToolBench (Xu et al., 2023) covers diverse software tools for single- and multi-step tasks but underplays long-horizon coordination. The Berkeley Function Calling Leaderboard (BFCL) (Patil et al., 2025) evaluates single, parallel, and multi-step calls, though plans remain shallow. τ-Bench (Yao et al., 2024) focuses on human–agent interaction under domain rules, highlighting alignment and information gathering more than proactive planning.

3 ML-TOOL-BENCH

Each task in ML-Tool-Bench, can be formalized as a Markov Decision Process (MDP) $(\mathcal{S}, \mathcal{A}, \mathcal{T}, R)$ (Puterman, 2014) The *state space* $\mathcal{S}$ consists of the entire interaction history: all AI, Human, and Tool messages together with artifacts such as dataframes and ML models. Whenever a tool is executed, its observations (e.g., outputs, errors, logs) are appended to the history and folded into the state, so that the state maintains an up-to-date record of both conversational and artifact changes. The initial state s_0 comprises the Kaggle challenge description along with the dataset.

The *action space* is defined as: $\mathcal{A} = \big(\mathcal{A}_{\text{tool}} \cup \{\varnothing\}\big) \times \big(\mathcal{A}_{\text{reason}} \cup \{\varnothing\}\big)$, where $\mathcal{A}_{\text{tool}}$ denotes the set of all tool invocations together with their full parameterizations (not just tool identity, but also

argument values and hyperparameters). This makes the benchmark challenging, since the effective size of $\mathcal{A}_{\text{tool}}$ can be very large rather than a small, discrete set. The set $\mathcal{A}_{\text{reason}}$ is the space of free-form reasoning steps, which we model as natural-language strings. The null element $\varnothing$ denotes "no action" in that component, allowing tool-only, reason-only, both, or neither at a step. Reasoning actions organize information, plan future steps, and inject prior knowledge; tool actions modify or analyze data and train/evaluate models, thereby updating the state's artifacts.

The *transition function* $\mathcal{T} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ maps a state–action pair (s, a) to the next state by appending the messages generated by the agent's action, appending tool messages (i.e., the observations produced), and updating artifacts accordingly.

The *reward function* R evaluates progress and can be instantiated in several ways: (i) an outcome reward granted upon successful challenge completion; (ii) a shaped reward providing intermediate credit for measurable progress; or (iii) an LLM-based evaluation of the current state, using the LLM as a judge (Zheng et al., 2023) and absolving us from providing the reward function.

3.1 SCRATCHPAD

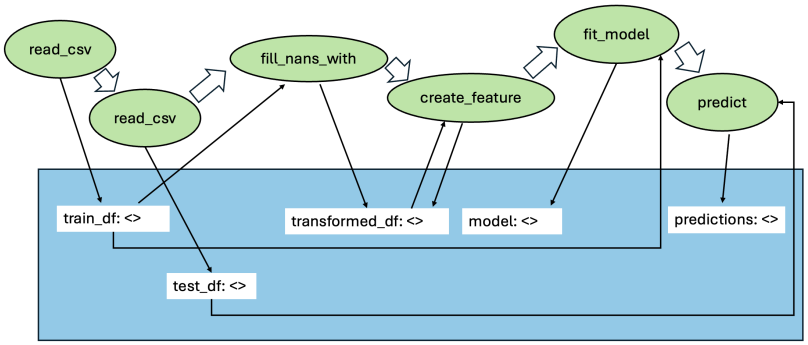

Figure 1: An illustration of our named-object management scheme. Green circles denote tool calls; the blue rectangle denotes the scratchpad (a key–value store). Each tool can read any named object from the scratchpad and write outputs back to it, depending on their read-write behavior. Arrows into a tool indicate inputs; arrows from a tool to the scratchpad indicate outputs. `read_csv` is a set tool; `fill_nans_with`, `fit_model`, and `predict` are get–set tools; `create_feature` is an override tool. There are two `read_csv` tool calls in the figure, one for train data and one for test.

Solving an ML challenge often involves storing large dataframes, models, and other complex artifacts as they cannot be directly passed as tool inputs by an LLM. A naive workaround is to maintain a single dataframe and model object that the agent incrementally modifies via tool calls. However, a single erroneous call can corrupt these objects, forcing a restart of the trajectory, and the agent becomes inflexible to create and reuse intermediate variables.

To address this, we adopt an in-memory, named-object management scheme: an agent assigns names to tool outputs, and tools accept references to named objects as inputs. Thus, agents can pass complex objects to tools by specifying the name under which the object is stored in the scratchpad. An illustration of this approach is presented in Figure 1. Implementing this requires modifying tools to operate on named references rather than raw objects; we describe these changes next.

3.2 TOOLS

We grant the agent access to a curated suite of 61 tools spanning data loading, data cleaning, feature engineering, and modeling. These tools are designed to be reasonably sufficient for solving tabular regression and classification tasks. Agent performance depends on the available toolset: in principle, a very large collection would maximize flexibility, but it results in an increased action space and complicates planning. We therefore adopt a fixed, compact tool set that trades some flexibility for a more tractable planning, while remaining adequate to solve the Kaggle challenges considered. For

modeling, we restrict to tree-based learners: Random Forest, XGBoost, LightGBM, and CatBoost, and linear/logistic regression, in light of the strong performance of tree-based methods on tabular Kaggle challenges (Grinsztajn et al., 2022). For more information on tools and how arbitrary user defined tools are modified to operate on named references rather than objects, refer to Appendix E

3.3 KAGGLE CHALLENGES

We select 15 tabular Kaggle ML challenges for ML-Tool-Bench: eight classification (binary and multiclass) and seven regression. These tasks are chosen so that they are solvable with our tool set. Several datasets are large (e.g., New York City Taxi Fare Prediction is $\sim$2.5 GB), so we randomly sample 10,000 data points from each competition's training set to keep planning computationally tractable. Because Kaggle test labels are hidden, we create an internal evaluation split by reserving 20% of the sampled training data as a test set with ground-truth labels. We evaluate using each competition's official metric and report agent performance as the corresponding public-leaderboard percentile. Our evaluation metric is chosen to accommodate a collection of regression and multi-class classification tasks. Note that Kaggle leaderboards are computed on a test set, the labels to which we do not have access to; our reported results are computed on our held-out test split. For more information on the Kaggle challenges, refer to Appendix C

4 APPROACHES

4.1 REACT

ReAct (Yao et al., 2023b) is a prompting framework that interleaves natural-language reasoning (*Thought*) with tool interaction (*Action*) and the subsequent *Observation* from the environment due to tool calling. ReAct augments the agent's action space to include the space of language, to account for thoughts or reasoning traces that do not affect the environment. Thoughts compose useful information from the current context and update the context to support future reasoning or actions. By explicitly exposing intermediate chain-of-thought alongside tool calls, ReAct enables agents to plan, invoke tools, and revise plans based on feedback. However, ReAct is unidirectional and can neglect potential alternative continuations from certain states, leading to locally optimal solutions (Zhuang et al., 2023; Zhou et al., 2024a).

4.2 MONTE CARLO TREE SEARCH (MCTS)

MCTS (Kocsis & Szepesvari, 2006) is a search algorithm that has achieved remarkable success in challenging domains such as Go (Silver et al., 2016) and Atari (Ye et al., 2021). MCTS builds a search tree where nodes correspond to states and edges correspond to actions. It comprises four phases: *selection*, *expansion*, *simulation/rollout*, and *backpropagation*. A common *selection* policy uses UCT (Upper Confidence Bound for Trees) (Kocsis & Szepesvari, 2006), choosing a child s of parent p such that: $s \in \arg\max_{s \in \mathbb{C}(p)} \; V(s) \; + \; w\sqrt{\ln N(p)/N(s)}$,

where $V(s)$ is the empirical value function, denoting the expected cumulative reward from state s, $N(p)$ is the parent's visit count, $N(s)$ is the child's visit count, $w > 0$ controls exploration, and $\mathbb{C}(p)$ denotes the set of children of p. Upon reaching a leaf node, it is *expanded* by selecting an action and adding the resulting next state as a child. From the newly expanded node, a *simulation* is run until the end of the episode or a fixed depth to obtain a reward r, which is then *backpropagated* along the trajectory to update values of all states along that trajectory: $V(s) \leftarrow (V(s)\,(N(s)-1) + r)/N(s)$. MCTS is well-suited to large, irregular action spaces and provides a principled trade-off between exploration and exploitation. A pictorial illustration of MCTS is provided in Appendix B.

4.3 LANGUAGE AGENT TREE SEARCH (LATS)

LATS (Zhou et al., 2024a) adapts MCTS to language agents by using LLMs both to propose actions (reasoning steps or tool calls) and to evaluate node values. At each expansion, the policy LLM suggests candidates, and an evaluator LLM scores partial trajectories based on estimated progress toward the task objective. The value of a state is taken to be a weighted average of the evaluator LLM's score and a self-consistency score (Wang et al., 2022), which upweights frequent candidates

in the expansion stage. In our tool-planning setting, we do not incorporate the self-consistency score into the value of a state. We observed that during the expansion phase, the LLM tends to propose only a small but distinct set of tool calls or reasoning steps, making the additional score unnecessary. LATS has shown improvements over purely reactive methods, such as ReAct (Yao et al., 2023b) on complex tasks. However, its value estimates can be noisy, and the effective planning depth may be limited by inconsistencies in evaluator scoring.

4.4 MCTS-SHAPED

In MCTS with shaped rewards, the agent receives intermediate credit for completing stages of the Kaggle ML challenge. The shaped-reward stages and their triggers are detailed below. Figure 2 provides an example to illustrate how rewards are provided in MCTS-Shaped.

Shaped-reward stages

1. **Train data loading**: reward when the agent successfully loads the training data.
2. **Test data loading**: reward when the agent successfully loads the test data. Note that test data does not have the target variable, that needs to be predicted.
3. **Combine train and test**: reward when the agent correctly concatenates train and test to enable consistent cleaning and feature engineering.
4. **Data cleaning**: reward when no missing values (NaNs) remain in the combined data.
5. **Feature engineering**: reward when (a) all categorical variables are properly encoded (e.g., one-hot or label encoding), and (b) the resulting feature dimensionality remains within a reasonable bound (to avoid exploding features from, e.g., high-cardinality text-like columns).
6. **Split back to train/test**: reward when the agent correctly splits the combined data back into train and test after transformations.
7. **Train features/target**: reward when the agent extracts $(X_{\text{train}}, y_{\text{train}})$ from the training dataframe using the correct target column.
8. **Test features**: reward when the agent extracts X_{test} from the test dataframe (which prior to this stage contains a dummy target), with correct arguments.
9. **Modeling**: reward when the agent successfully fits a model on the training data; the reward is proportional to cross-validation performance.
10. **Create submission**: reward when the agent generates predictions on the test data and writes a valid submission CSV to disk.

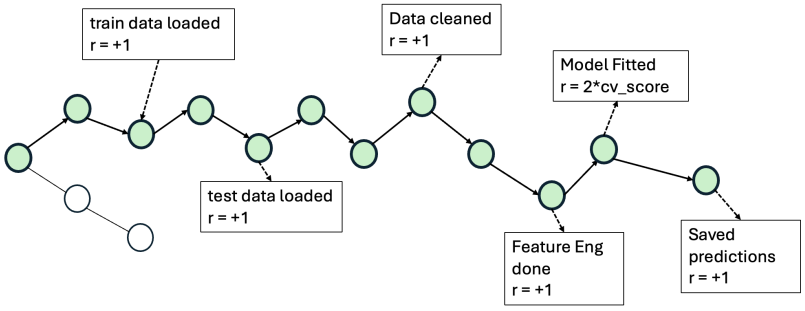

Figure 2: An example illustration of how rewards are provided in MCTS-Shaped. If a particular stage is judged to be successfully completed at a node, a reward is given, which is used to update the value of all the nodes in this trajectory. It needs to be noted that these stage-wise rewards are only provided once per trajectory and only if the earlier stages were successfully completed.

It needs to be noted that all of the stage rewards are provided to the agent only once per trajectory, and only if the earlier stages were successfully completed. The provided stage rewards are used to update the value of all the nodes in the trajectory. We verify stage completion using a reward function that inspects the node scratchpad and tool messages, confirming (i) that artifacts satisfy required properties (e.g., no NaNs for data cleaning; all columns encoded for feature engineering) and (ii) that the correct tools were invoked as evidenced by the tool logs.

4.5 HIERARCHICAL MCTS

We propose Hierarchical MCTS to improve over ReAct (Yao et al., 2023b), LATS (Zhou et al., 2024a), and classical MCTS (Kocsis & Szepesvari, 2006) in generating performant tool-use trajectories for solving Kaggle challenges within ML-Tool-Bench. Hierarchical MCTS decomposes a complex task into an ordered sequence of subtasks. We partition the available tools and assign them to relevant sub-tasks manually. For each subtask, MCTS searches its local state–action space to identify solution nodes. The solution nodes from one subtask are appended to the root of the next subtask, and the search continues. To avoid being trapped in locally optimal (but globally suboptimal) choices, we enumerate all solution nodes within each subtask up to a prescribed maximum subtask search depth. If there are no solution nodes identified after a subtask, the search terminates and we return 'No Solution Found'. The solution node with the highest value, at the final subtask, is returned as the solution of the Hierarchical MCTS search. Importantly, the agent is given only the tools relevant to the current subtask (tool masking), which reduces the branching factor and focuses the search. Figure 3 illustrates the overall procedure. Hierarchical MCTS is similar to the options framework (Sutton et al., 1999), that break down a complex problem into a hierarchy of sub-tasks, making the learning process more efficient and manageable for an agent.

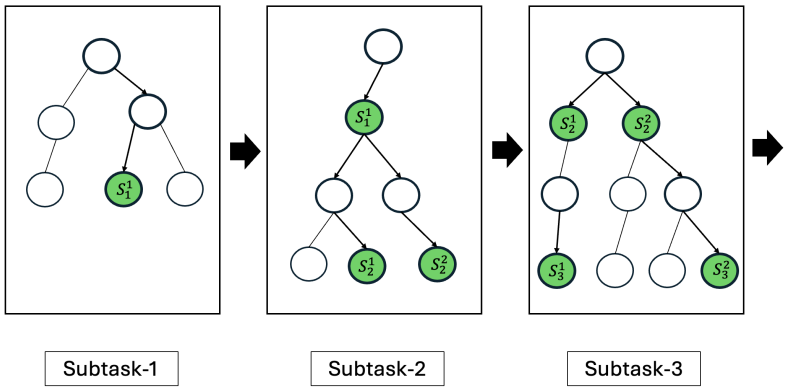

Figure 3: A schematic of Hierarchical MCTS. The task is decomposed into an ordered sequence of subtasks. For each subtask, MCTS searches for all solution nodes up to a prescribed maximum subtask depth to avoid locally optimal but globally suboptimal choices. The solution nodes from subtask t are appended to the root of subtask $t+1$, and the search resumes. In the example, the solution node from subtask 1, S_1^1, initializes subtask 2; its solution nodes S_2^1 and S_2^2 initialize subtask 3, and so on. The highest-value solution at the final subtask is returned as the overall outcome of Hierarchical MCTS.

5 EXPERIMENTS

We evaluate the tool-planning performance of two language models—GPT-4o and GPT-4.1-mini, on ML-Tool-Bench. For each model, we compare five planning algorithms: (i) *ReAct* (Yao et al., 2023b); (ii) *LATS* (Zhou et al., 2024a); (iii) Monte Carlo Tree Search (MCTS) with outcome-based rewards, where the agent is rewarded upon successfully training a model or producing a valid submission file (denoted *MCTS-Outcome*); (iv) MCTS with shaped rewards, where the agent receives intermediate credit for completing stages of the Kaggle ML workflow (denoted *MCTS-Shaped*); and (v) *Hierarchical MCTS*: the Kaggle challenge is decomposed into subtasks. We use the reward stages defined for *MCTS-Shaped* as subtasks. A node is a solution node for a subtask, if it satisfies the reward condition for the stage corresponding to that subtask.

5.1 IMPLEMENTATION DETAILS

When using tree-search methods with our in-memory, named-object scheme, we adopt a *path-local scratchpad*, where each node v contains a scratchpad $\mathcal{S}(v)$, that stores only the objects produced by

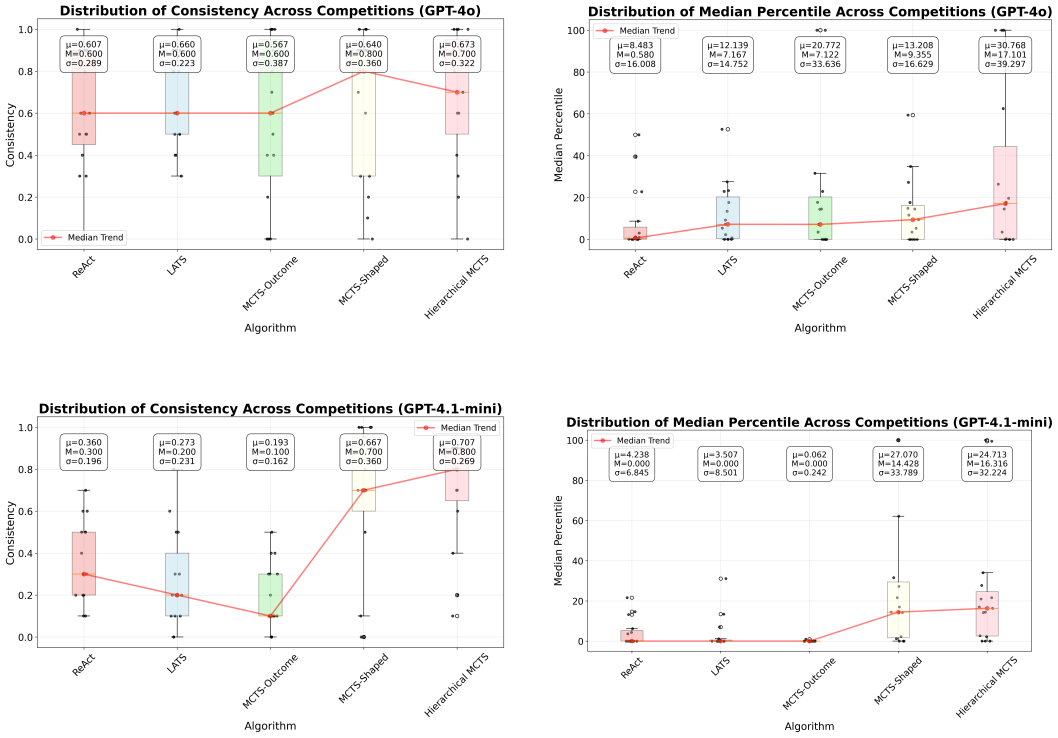

Figure 4: Plots of consistency and median leaderboard percentile across all competitions in ML-Tool-Bench, for different planning algorithms. The top row shows results for GPT-4o, with the left plot showing consistency and the right plot showing the median leaderboard percentile. The bottom row shows results for GPT-4.1-mini. Hierarchical MCTS outperforms LATS and ReAct, followed by MCTS-Shaped, in terms of leaderboard performance, for both LLMs. Also, both Hierarchical MCTS and MCTS-Shaped improve consistency over the other baselines. In the box plots, μ denotes the mean, σ denotes the standard deviation, and M denotes the Median

the tool call at that node. During expansion, the LLM proposes candidate actions. For a candidate that is a tool call, the accessible memory is the path union: $\mathcal{S}^*(v) = \bigcup_{u \in \text{path}(\text{root} \to v)} \mathcal{S}(u)$, and the LLM may reference any named object in $\mathcal{S}^*(v)$ as tool arguments. The tool's outputs are written to the child's scratchpad $\mathcal{S}(\text{child})$, preserving isolation per node while enabling reuse of intermediate artifacts along the trajectory.

LATS: To estimate the value of a state, we provide an evaluator LLM with all `AIMessage` and `ToolMessage` entries along the path from the root to the current node; it scores the trajectory by the progress made toward solving the Kaggle challenge. To propose candidate actions, we similarly pass the full trajectory history to the LLM, which returns new reasoning steps or tool calls. Unlike the original LATS formulation, we omit a self-consistency score from the value estimate, as at each expansion the agent typically proposes a small number of distinct candidates.

MCTS: We propose new candidate nodes during the expansion phase using the same approach listed in LATS. To evaluate the value of a node, we check if it produces a model or a valid submission file in the outcome rewards case. In the shaped rewards case, a node is provided a reward if it successfully completes a stage, as detailed earlier. In the case of Hierarchical MCTS, we designate a node as a solution node of the subtask, if it successfully completes the stage corresponding to that subtask. Additionally, across all MCTS variants, we apply a per-level depth penalty of 0.1 to discourage unnecessarily long trajectories that fail to make progress toward the goal.

In addition to rewards, we provide targeted textual feedback to help the agent refine its plan. When a stage fails, the agent receives an explanation of the failure. For example, in *feature engineering* we flag remaining categorical columns or an excessive increase in feature dimensionality; in *data*

cleaning we report the presence of missing values. If a tool invocation fails, we return an explicit message along with the tool's docstring to guide correct usage on the next attempt. We find that such feedback is crucial for consistently producing valid trajectories. This textual feedback is provided for all the MCTS variants (*MCTS-Outcome*, *MCTS-Shaped*, and *Hierarchical-MCTS*).

Ideally, we would run Monte Carlo rollouts to a fixed depth or until episode termination and use the return to update the value of all the nodes in the trajectory. Running to termination is impractical due to cost and compute constraints. Shallow rollouts (depth 3–5) are viable but GPT usage across many Kaggle challenges, planning algorithms, and trials, and roll outs at each state, resulted in extremely high costs and was infeasible. Learning value functions to approximate the value of states (Silver et al., 2016) is also not straightforward, on account of complex artifacts that are a part of the state. Consequently, we use the immediate reward at the current state (a depth-0 rollout), yielding a best-first search with a UCT-style exploration bonus. When budget permits, using small depth rollouts is preferred.

5.2 Results

We evaluate GPT-4o and GPT-4.1-mini on our benchmark. For each algorithm–Kaggle challenge combination, we run 10 trials. We define *consistency* as the proportion of valid trajectories (e.g., 4 valid trajectories out of 10 trials yields a consistency of 0.4). For each trial, we evaluate predictions against the provided test labels using the competition's official metric and compare against the leaderboard to obtain a leaderboard percentile. For each algorithm and competition, we report the median percentile across the 10 trials. Figure 4 presents boxplots for all algorithms, summarizing the distribution of leaderboard percentiles across all competitions in our benchmark. For further details on consistency and leaderboard percentiles for both models, refer to Appendix D.

As shown in Figure 4, Hierarchical MCTS improves leaderboard performance compared to ReAct, LATS, and MCTS-Outcome, followed by MCTS-Shaped, for both GPT-4o and GPT-4.1-mini. Moreover, both Hierarchical MCTS and MCTS-Shaped achieve higher consistency than the other baselines. For GPT-4o, Hierarchical MCTS shows improvement over LATS by 9.93 percentile positions on the leaderboard and over ReAct by 16.52 percentile positions, taking the median across all competitions. For GPT-4.1-mini, Hierarchical MCTS improved over MCTS-Shaped by 1.89 percentile positions on the leaderboard, while both ReAct and LATS had a median leaderboard percentile position of 0 across all competitions. These results highlight that as toolsets become more complex and larger, it is important either to introduce hierarchy—decomposing the original task into subtasks with corresponding reward functions, or to employ shaped rewards that guide the search toward solutions. In contrast, unidirectional planning strategies like ReAct do not perform well. Similarly, tree-search methods such as LATS, that rely solely on LLM evaluation also fail, as LLMs provide inconsistent scores to nodes when trajectory lengths increase, due to the accumulation of messages and artifacts that must be considered during evaluation.

6 Conclusion

We introduced ML-Tool-Bench, a benchmark for evaluating the planning capabilities of tool-augmented LLMs on tabular Kaggle challenges. Existing tool-use benchmarks (Xu et al., 2023; Patil et al., 2025; Yao et al., 2024) primarily assess tool selection and argument grounding, rather than long-horizon planning. By contrast, many ML agents generate code directly; while flexible, this approach sacrifices modularity, reliability, and safety compared to operating within a curated toolset. Empirically, we found that ReAct and LATS struggle to consistently produce valid and performant trajectories. We proposed two improved approaches: (i) MCTS with shaped, deterministic rewards, and (ii) Hierarchical MCTS, which decomposes problems into sequenced subtasks. Across two models, Hierarchical MCTS achieved the best leaderboard performance compared to other baselines, while both Hierarchical MCTS and MCTS-Shaped improved consistency, measured as the fraction of valid trajectories. These results suggest that incorporating subtask decomposition with deterministic rewards, rather than relying on subjective LLM evaluation, yields performance gains as the set of available tools grows in size and complexity.

REFERENCES

Altaf Allah Abbassi, Leuson Da Silva, Amin Nikanjam, and Foutse Khomh. A taxonomy of inefficiencies in llm-generated python code, 2025. URL https://arxiv.org/abs/2503.06327.

Rogerio Bonatti, Dan Zhao, Francesco Bonacci, Dillon Dupont, Sara Abdali, Yinheng Li, Yadong Lu, Justin Wagle, Kazuhito Koishida, Arthur Bucker, Lawrence Jang, and Zack Hui. Windows agent arena: Evaluating multi-modal os agents at scale, 2024. URL https://arxiv.org/abs/2409.08264.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Lilian Weng, and Aleksander Madry. Mle-bench: Evaluating machine learning agents on machine learning engineering, 2025. URL https://arxiv.org/abs/2410.07095.

Yongchao Chen, Harsh Jhamtani, Srinagesh Sharma, Chuchu Fan, and Chi Wang. Steering large language models between code execution and textual reasoning, 2025. URL https://arxiv.org/abs/2410.03524.

Jiazhan Feng, Shijue Huang, Xingwei Qu, Ge Zhang, Yujia Qin, Baoquan Zhong, Chengquan Jiang, Jinxin Chi, and Wanjun Zhong. Retool: Reinforcement learning for strategic tool use in llms. *ArXiv*, abs/2504.11536, 2025. URL https://api.semanticscholar.org/CorpusID:277824366.

Xidong Feng, Ziyu Wan, Muning Wen, Stephen Marcus McAleer, Ying Wen, Weinan Zhang, and Jun Wang. Alphazero-like tree-search can guide large language model decoding and training, 2024. URL https://arxiv.org/abs/2309.17179.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Minlie Huang, Nan Duan, and Weizhu Chen. Tora: A tool-integrated reasoning agent for mathematical problem solving. *ArXiv*, abs/2309.17452, 2023. URL https://api.semanticscholar.org/CorpusID:263310365.

Léo Grinsztajn, Edouard Oyallon, and Gaël Varoquaux. Why do tree-based models still outperform deep learning on tabular data?, 2022. URL https://arxiv.org/abs/2207.08815.

Antoine Grosnit, Alexandre Maraval, James Doran, Giuseppe Paolo, Albert Thomas, Refinath Shahul Hameed Nabeezath Beevi, Jonas Gonzalez, Khyati Khandelwal, Ignacio Iacobacci, Abdelhakim Benechehab, Hamza Cherkaoui, Youssef Attia El-Hili, Kun Shao, Jianye Hao, Jun Yao, Balazs Kegl, Haitham Bou-Ammar, and Jun Wang. Large language models orchestrating structured reasoning achieve kaggle grandmaster level, 2024. URL https://arxiv.org/abs/2411.03562.

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.

Qian Huang, Jian Vora, Percy Liang, and Jure Leskovec. Mlagentbench: Evaluating language agents on machine learning experimentation, 2024. URL https://arxiv.org/abs/2310.03302.

Zhengyao Jiang, Dominik Schmidt, Dhruv Srikanth, Dixing Xu, Ian Kaplan, Deniss Jacenko, and Yuxiang Wu. Aide: Ai-driven exploration in the space of code, 2025. URL https://arxiv.org/abs/2502.13138.

Liqiang Jing, Zhehui Huang, Xiaoyang Wang, Wenlin Yao, Wenhao Yu, Kaixin Ma, Hongming Zhang, Xinya Du, and Dong Yu. Dsbench: How far are data science agents from becoming data science experts?, 2025. URL https://arxiv.org/abs/2409.07703.

Levente Kocsis and Csaba Szepesvari. Bandit based monte-carlo planning. In *European Conference on Machine Learning*, 2006. URL https://api.semanticscholar.org/CorpusID:15184765.

Xuefeng Li, Haoyang Zou, and Pengfei Liu. Torl: Scaling tool-integrated rl. *ArXiv*, abs/2503.23383, 2025. URL https://api.semanticscholar.org/CorpusID:277451754.

Changshu Liu, Yang Chen, and Reyhaneh Jabbarvand. Codemind: Evaluating large language models for code reasoning, 2025. URL https://arxiv.org/abs/2402.09664.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive apis, 2023. URL https://arxiv.org/abs/2305.15334.

Shishir G. Patil, Huanzhi Mao, Charlie Cheng-Jie Ji, Fanjia Yan, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. The berkeley function calling leaderboard (bfcl): From tool use to agentic evaluation of large language models. In *Forty-second International Conference on Machine Learning*, 2025.

Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

Rushi Qiang, Yuchen Zhuang, Yinghao Li, Dingu Sagar V K, Rongzhi Zhang, Changhao Li, Ian Shu-Hei Wong, Sherry Yang, Percy Liang, Chao Zhang, and Bo Dai. Mle-dojo: Interactive environments for empowering llm agents in machine learning engineering, 2025. URL https://arxiv.org/abs/2505.07782.

Yujia Qin, Shi Liang, Yining Ye, Kunlun Zhu, Lan Yan, Ya-Ting Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Runchu Tian, Ruobing Xie, Jie Zhou, Marc H. Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. Toolllm: Facilitating large language models to master 16000+ real-world apis. *ArXiv*, abs/2307.16789, 2023. URL https://api.semanticscholar.org/CorpusID:260334759.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *ArXiv*, abs/2302.04761, 2023. URL https://api.semanticscholar.org/CorpusID:256697342.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

Joykirat Singh, Raghav Magazine, Yash Pandya, and Akshay Nambi. Agentic reasoning and tool integration for llms via reinforcement learning, 2025. URL https://arxiv.org/abs/2505.01441.

Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988. doi: 10.1007/BF00115009.

Richard S Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211, 1999.

Edan Toledo, Karen Hambardzumyan, Martin Josifoski, Rishi Hazra, Nicolas Baldwin, Alexis Audran-Reiss, Michael Kuchnik, Despoina Magka, Minqi Jiang, Alisia Maria Lupidi, Andrei Lupu, Roberta Raileanu, Kelvin Niu, Tatiana Shavrina, Jean-Christophe Gagnon-Audet, Michael Shvartsman, Shagun Sodhani, Alexander H. Miller, Abhishek Charnalia, Derek Dunfield, Carole-Jean Wu, Pontus Stenetorp, Nicola Cancedda, Jakob Nicolaus Foerster, and Yoram Bachrach. Ai research agents for machine learning: Search, exploration, and generalization in mle-bench, 2025. URL https://arxiv.org/abs/2507.02554.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

Qinzhuo Wu, Wei Liu, Jian Luan, and Bin Wang. Toolplanner: A tool augmented llm for multi granularity instructions with path planning and feedback. *ArXiv*, abs/2409.14826, 2024. URL https://api.semanticscholar.org/CorpusID:272827086.

Qiantong Xu, Fenglu Hong, Bo Li, Changran Hu, Zhengyu Chen, and Jian Zhang. On the tool manipulation capability of open-source large language models, 2023. URL https://arxiv.org/abs/2305.16504.

Sherry Yang, Joy He-Yueya, and Percy Liang. Reinforcement learning for machine learning engineering agents. 2025. URL https://api.semanticscholar.org/CorpusID:281080955.

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. *ArXiv*, abs/2305.10601, 2023a. URL https://api.semanticscholar.org/CorpusID:258762525.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023b. URL https://arxiv.org/abs/2210.03629.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains, 2024. URL https://arxiv.org/abs/2406.12045.

Weirui Ye, Shaohuai Liu, Thanard Kurutach, Pieter Abbeel, and Yang Gao. Mastering atari games with limited data. *Advances in neural information processing systems*, 34:25476–25488, 2021.

Yuanqing Yu, Zhefan Wang, Weizhi Ma, Zhicheng Guo, Jingtao Zhan, Shuai Wang, Chuhan Wu, Zhiqiang Guo, and Min Zhang. Steptool: Enhancing multi-step tool usage in llms via step-grained reinforcement learning. 2024. URL https://api.semanticscholar.org/CorpusID:273233670.

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.

Dan Zhang, Sining Zhoubian, Min Cai, Fengzu Li, Lekang Yang, Wei Wang, Tianjiao Dong, Ziniu Hu, Jie Tang, and Yisong Yue. Datascibench: An llm agent benchmark for data science, 2025. URL https://arxiv.org/abs/2502.13897.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena, 2023. URL https://arxiv.org/abs/2306.05685.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning acting and planning in language models, 2024a. URL https://arxiv.org/abs/2310.04406.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. Webarena: A realistic web environment for building autonomous agents, 2024b. URL https://arxiv.org/abs/2307.13854.

Yuchen Zhuang, Xiang Chen, Tong Yu, Saayan Mitra, Victor S. Bursztyn, Ryan A. Rossi, Somdeb Sarkhel, and Chao Zhang. Toolchain*: Efficient action space navigation in large language models with a* search. *ArXiv*, abs/2310.13227, 2023. URL https://api.semanticscholar.org/CorpusID:264405734.

A Large Language Model Usage

Large Language Models (LLMs) were used for grammatical editing and improving writing flow. Additionally, LLMs assisted the authors in conducting literature surveys and identifying related work. LLMs were also used to aid in developing the ML-Tool-Bench toolset. LLMs were used

to assist with code generation, debugging, and documentation for components of the ML-Tool-Bench toolset, based on tool descriptions provided by the authors. LLMs were also used to assign relevant tools to each subtask in the proposed Hierarchical MCTS approach. All implementations were reviewed and validated by the authors. All research methodology, experimental design, data analysis, and scientific conclusions are entirely the work of the human authors.

B Approaches

B.1 Monte Carlo Tree Search

Figure 5 provides a pictorial illustration of the MCTS algorithm.

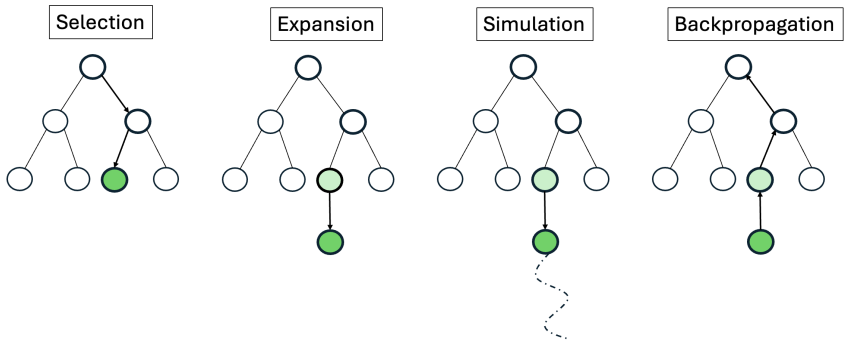

Figure 5: A pictorial illustration of Monte Carlo Tree Search

C Kaggle Challenges

The list of Kaggle challenges present in ML-Tool-Bench, and the corresponding ML problem types of each challenge are presented in Table 1

Challenge	**Type**
Santander Value Prediction Challenge	Regression
New York City Taxi Fare Prediction	Regression
New York City Taxi Trip Duration	Regression
Predicting the Beats-per-Minute of Songs	Regression
Predict Calorie Expenditure	Regression
Regression with a Tabular California Housing Dataset	Regression
Regression of Used Car Prices	Regression
Porto Seguro Safe Driver Prediction	Binary Classification
Costa Rican Household Poverty Prediction	Multi-Class Classification
Forest Cover Type (Kernels Only)	Multi-Class Classification
Santander Customer Transaction Prediction	Binary Classification
Binary Prediction of Poisonous Mushrooms	Binary Classification
Spaceship Titanic	Binary Classification
Binary Classification with a Bank Dataset	Binary Classification
Binary Classification with a Bank Churn Dataset	Binary Classification

Table 1: Kaggle challenges used in ML-Tool-Bench with problem type.

D Results

In this section, we provide the exact consistency and performance values for each of the 15 challenges and the two models (GPT-4o and GPT-4.1-mini). Tables 2 and 3 show the consistency and

Competition	ReAct	LATS	MCTS-Outcome	MCTS-Shaped	Hierarchical MCTS
Spaceship Titanic	0.6	0.4	**0.9**	0.8	0.6
Santander Value Prediction Challenge	0.9	0.6	0.5	**1**	**1**
NYC Taxi Fare Prediction	0	0.5	0.2	0.3	**0.9**
NYC Taxi Trip Duration	0.3	0.3	**1**	0.2	0.3
BPM Prediction	**1**	0.6	0	0.9	0.7
Calorie Expenditure Prediction	0.8	0.8	**1**	0.9	**1**
california Housing Regression	0.9	0.9	**1**	0.9	0.9
Used Car Prices Regression	0.9	0.4	0.4	0.9	**1**
Porto Seguro Safe Driver Prediction	0.3	**0.5**	0	0.1	0.2
Costa Rican Household Poverty Level Prediction	0.5	0.5	0.7	0.3	**1**
Forest Cover Type Prediction	0.6	**0.9**	0	0	0
Santander Customer Transaction Prediction	0.5	**0.8**	**0.8**	0.7	0.4
Poisonous Mushroom Prediction	0.9	**1**	**1**	**1**	0.8
Bank Deposit Classification	0.5	**0.8**	0.4	0.6	0.7
Bank Churn Classification	0.4	0.9	0.6	**1**	0.6
Overall (Median)	0.6	0.6	0.6	**0.8**	0.7

Table 2: Consistency across 15 competitions for five planning algorithms for GPT-4o.

Competition	ReAct	LATS	MCTS-Outcome	MCTS-Shaped	Hierarchical MCTS
Spaceship Titanic	39.54	0	22.88	59.44	**62.55**
Santander Value Prediction Challenge	0.09	7.17	7.12	**14.87**	0.27
NYC Taxi Fare Prediction	0	9.23	0	0	**19.66**
NYC Taxi Trip Duration	0	0	**100.0**	0	0
BPM Prediction	0.51	52.63	0	5.26	**100**
Calorie Expenditure Prediction	0.16	13.43	**14.47**	**14.47**	**14.47**
california Housing Regression	0.58	0.65	14.35	11.59	**17.10**
Used Car Prices Regression	3.0	0	0	9.35	**100**
Porto Seguro Safe Driver Prediction	0	**5.34**	0	0	0
Costa Rican Household Poverty Level Prediction	50	0	**100**	0	**100**
Forest Cover Type Prediction	0.84	**23.32**	0	0	0
Santander Customer Transaction Prediction	1.14	2.27	**3.49**	**3.49**	0
Poisonous Mushroom Prediction	**22.75**	17.61	17.69	17.62	17.62
Bank Deposit Classification	8.64	**27.50**	0	27.24	26.37
Bank Churn Classification	0	22.93	31.57	**34.79**	3.47
Overall (Median)	0.58	7.17	7.12	9.36	**17.10**

Table 3: Median Leaderboard percentile across 15 competitions for five planning algorithms for GPT-4o.

leaderboard percentiles for all algorithms across all competitions in ML-Tool-Bench, for GPT-4o. Similarly, Tables 4 and 5 show the consistency and leaderboard percentiles for all algorithms across all competitions in ML-Tool-Bench, for GPT-4.1-mini.

Competition	ReAct	LATS	MCTS-Outcome	MCTS-Shaped	Hierarchical MCTS
Spaceship Titanic	0.1	0	0.4	**0.7**	0.2
Santander Value Prediction Challenge	0.6	0.2	0	**0.9**	**0.9**
NYC Taxi Fare Prediction	0.2	0	0.1	0.1	**0.9**
NYC Taxi Trip Duration	0.1	0.1	0.3	0.5	**0.9**
BPM Prediction	0.4	0.1	0	**1**	0.8
Calorie Expenditure Prediction	0.6	0.6	0	**0.9**	**0.9**
california Housing Regression	0.7	0.3	0.3	**0.8**	**0.8**
Used Car Prices Regression	0.3	0.2	0.4	**0.7**	0.4
Porto Seguro Safe Driver Prediction	0.2	0.1	0.1	0.7	**0.9**
Costa Rican Household Poverty Level Prediction	**0.2**	**0.2**	0.1	0	0.1
Forest Cover Type Prediction	0.2	0.2	0.3	0	**0.7**
Santander Customer Transaction Prediction	0.3	0.5	0.1	**0.7**	**0.7**
Poisonous Mushroom Prediction	0.5	0.5	0.2	**1**	0.6
Bank Deposit Classification	0.5	0.3	0.1	**1**	0.9
Bank Churn Classification	0.5	0.8	0.5	**1**	0.9
Overall (Median)	0.3	0.2	0.1	0.7	**0.8**

Table 4: Consistency across 15 competitions for five planning algorithms for GPT-4.1-mini.

Competition	ReAct	LATS	MCTS-Outcome	MCTS-Shaped	Hierarchical MCTS
Spaceship Titanic	0	0	0	**62.11**	0
Santander Value Prediction Challenge	3.60	0	0	**14.24**	**14.24**
NYC Taxi Fare Prediction	0	0	0	0	**20.94**
NYC Taxi Trip Duration	0	0	0	1.24	**2.70**
BPM Prediction	0	0	0	**100**	**100**
Calorie Expenditure Prediction	4.31	13.41	0	**14.43**	**14.43**
california Housing Regression	**21.59**	0	0	**21.59**	**21.59**
Used Car Prices Regression	0	0	0	**100**	0
Porto Seguro Safe Driver Prediction	0	0	0	**17.01**	**17.01**
Costa Rican Household Poverty Level Prediction	0	0	0	0	0
Forest Cover Type Prediction	0	0	0	0	**99.44**
Santander Customer Transaction Prediction	0	1.17	0	**2.27**	**2.27**
Poisonous Mushroom Prediction	6.23	6.97	0	14.36	**16.32**
Bank Deposit Classification	13.19	0	0	27.24	**27.73**
Bank Churn Classification	14.66	31.09	0.94	31.57	**34.02**
Overall (Median)	0	0	0	14.43	**16.32**

Table 5: Median Leaderboard percentile across 15 competitions for five planning algorithms for GPT-4.1-mini.

Stage	Number of Tools
Data Loading	6
Data Cleaning	9
Feature Engineering	30
Modeling	10
Evaluation/Prediction	10

Table 6: Number of tools available at each stage of a Kaggle-style workflow. In total, 61 tools are provided spanning data loading, cleaning, feature engineering, and modeling. Some tools can appear in more than one stage

E TOOLS

In this section, we describe the various tools that are part of ML-Tool-Bench. Table 6 shows the number of tools in our toolset that are part of each stage in solving an ML challenge on Kaggle. Table 7 provides info about all the tools in the curated toolset provided by ML-Tool-Bench

Decorators for named references To enable tools to operate on named references rather than raw objects, we design four decorators that adapt arbitrary user-provided functions to our scratchpad interface according to their read–write behavior. We categorize tools into four types:

1. **Set tool**: saves an object to memory. Example: `read_csv` loads a dataframe and stores it under a provided name.
2. **Get tool**: reads an object from memory. Example: `get_dataframe_summary` loads a dataframe and returns a brief textual summary to guide subsequent planning.
3. **Get–Set tool**: reads an object from memory and writes a new object to memory. Example: `fit_randomforest_model` takes as input, a dataframe, and returns a fitted model.
4. **Override tool**: reads an object, returns an updated object, and overwrites the input variable binding with the returned value. Example: `cast_column` loads a dataframe and returns a modified dataframe that replaces the original.

Accordingly, we provide four decorators: `make_get_tool`, `make_set_tool`, `make_get_and_set_tool`, and `make_override_tool`, that automatically wrap user-provided tools to operate on named references and integrate with the scratchpad.

Function Signature	Description
Modeling Functions	

`fit_logistic_regressor(X_train, y_train, cv=5)`	Fit Logistic Regression model
`fit_linear_regressor(X_train, y_train, cv=5)`	Fit Linear Regression model
`fit_random_forest_regressor(X_train, y_train, cv=5)`	Fit Random Forest Regressor
`fit_random_forest_classifier(X_train, y_train, cv=5)`	Fit Random Forest Classifier
`fit_xgboost_regressor(X_train, y_train, cv=5)`	Fit XGBoost Regressor
`fit_xgboost_classifier(X_train, y_train, cv=5)`	Fit XGBoost Classifier
`fit_lightgbm_regressor(X_train, y_train, cv=5)`	Fit LightGBM Regressor
`fit_lightgbm_classifier(X_train, y_train, cv=5)`	Fit LightGBM Classifier
`fit_catboost_regressor(X_train, y_train, cv=5)`	Fit CatBoost Regressor
`fit_catboost_classifier(X_train, y_train, cv=5)`	Fit CatBoost Classifier
Data Loading Functions	
`read_data(filepath)`	Read CSV data into a pandas DataFrame
Feature Engineering and Functions to get Dataframe information	
`create_numeric_feature(df, name, expression)`	Create a numeric feature using a pandas expression
`create_categorical_feature(df, name, source_column, mapping)`	Create a categorical feature by mapping values from a source column
`create_conditional_feature(df, name, condition, true_value, false_value)`	Create a feature based on a condition
`extract_string_pattern(df, name, source_column, pattern, group=0)`	Extract pattern from string column using regex
`split_string_column(df, name_prefix, source_column, delimiter, max_splits=-1, indices=None)`	Split string column and create separate features
`create_group_aggregation(df, name, group_column, agg_column, agg_func)`	Create feature by aggregating within groups
`get_group_aggregation(df, group_column, agg_column, agg_func)`	Get aggregation result without adding it to the DataFrame
`create_rolling_feature(df, name, source_column, window, agg_func='mean')`	Create rolling window feature
`create_lag_feature(df, name, source_column, lag=1)`	Create lagged feature
`create_lead_feature(df, name, source_column, lead=1)`	Create leading feature
`extract_datetime_features(df, datetime_column, features=None)`	Extract datetime features from datetime column
`create_time_delta(df, name, start_column, end_column, unit='D')`	Create time delta feature between two datetime columns
`apply_custom_function(df, name, source_columns, func)`	Apply custom function to create feature
`fillna_with_value(df, columns, value)`	Fill missing values with a specific value
`fillna_with_median(df, columns=None)`	Fill missing values with median of the column
`fillna_with_mean(df, columns=None)`	Fill missing values with mean of the column

`fillna_with_mode(df, columns=None)`	Fill missing values with mode of the column
`fillna_with_condition(df, target_column, condition, fill_value)`	Fill missing values in a column based on a condition
`fillna_with_multiple_conditions(df, target_column, conditions_and_values)`	Fill missing values in a column based on multiple conditions
`fillna_with_conditional_aggregation(df, target_column, condition_column, condition_values, agg_func='mean')`	Fill missing values using conditional aggregation based on another column's values
`fillna_with_custom_function(df, target_column, condition, custom_func)`	Fill missing values using a custom function based on a condition
`drop_rows_with_missing(df, columns=None, threshold=None)`	Drop rows with missing values
`get_missing_summary(df)`	Get a summary of missing values in the DataFrame
`cast_columns(df, column_type_mapping)`	Cast columns to specified data types
`cast_numeric_columns(df, columns=None, target_type='float')`	Cast numeric columns to specified type
`cast_integer_columns_to_float(df, columns=None)`	Cast integer columns to float type
`cast_categorical_columns(df, columns=None)`	Cast categorical columns to category type
`one_hot_encode(df, columns=None, drop_first=True, prefix=None)`	One-hot encode categorical columns
`label_encode(df, columns=None)`	Label encode categorical columns
`normalize_features(df, columns=None, method='standard')`	Normalize numeric features
`encode_all_categorical_columns(df, method='one_hot', drop_first=True)`	Encode all categorical/object columns using specified method
`normalize_all_numerical_columns(df, method='standard')`	Normalize all numerical columns using specified method
`concatenate_train_test(train_df, test_df)`	Concatenate train and test data with tracking columns for proper splitting
`split_combined_into_train_test(combined)`	Split combined data back into train and test using tracking columns
`convert_dataframe_to_features_target(df, target_column, is_train=True)`	Convert DataFrame to features and target format
`convert_to_dataframe(data, **kwargs)`	Convert various data types to pandas DataFrame
`drop_feature(df, column)`	Drop feature(s) from the DataFrame
`get_features(df, columns)`	Extract specific features (columns) from the DataFrame
`concatenate_dataframes(df1, df2, axis=0)`	Concatenate two DataFrames
`join_dataframes(left_df, right_df, left_on, right_on=None, how='inner', suffixes=('_x', '_y'))`	Join two DataFrames using pandas merge functionality
`rename_feature(df, old_name, new_name)`	Rename feature(s)
`get_unique_values(df, column, sort=True, include_counts=True)`	Get unique values from a column as a DataFrame
`get_dataframe_dtypes_summary(df)`	Get comprehensive summary of the dtypes in the entire DataFrame
`filter_dataframe(df, condition)`	Filter DataFrame using a boolean condition
Model Utilities	
`save_model(model, filepath='model.pkl')`	Save the trained model to disk using pickle

`load_model(filepath)`	Load a trained model from disk using pickle
`save_dataframe_to_csv(df, filepath)`	Save a DataFrame to CSV file
Model Evaluation Functions	
`evaluate_regression_model(model, X_test, y_test, model_name="model", eval_data_label='test')`	Evaluate a trained regression model on data
`evaluate_classification_model(model, X_test, y_test, model_name="model", eval_data_label='test')`	Evaluate a trained classification model on data
`predict_target(model, X_data, model_name="model", return_probabilities=False)`	Make predictions using a trained model

Table 7: All tools in the curated toolset provided by ML-Tool-Bench.