Google Cloud

MLE-STAR: Machine Learning Engineering Agent via Search and Targeted Refinement

Jaehyun Nam[1 2 *], **Jinsung Yoon**[1], **Jiefeng Chen**[1], **Jinwoo Shin**[2], **Sercan Ö. Arık**[1] and **Tomas Pfister**[1]
[1]Google Cloud, [2]KAIST

Agents based on large language models (LLMs) for machine learning engineering (MLE) can automatically implement ML models via code generation. However, existing approaches to build such agents often rely heavily on inherent LLM knowledge and employ coarse exploration strategies that modify the entire code structure at once. This limits their ability to select effective task-specific models and perform deep exploration within specific components, such as experimenting extensively with feature engineering options. To overcome these, we propose *MLE-STAR*, a novel approach to build MLE agents. MLE-STAR first leverages external knowledge by using a search engine to retrieve effective models from the web, forming an initial solution, then iteratively refines it by exploring various strategies targeting specific ML components. This exploration is guided by ablation studies analyzing the impact of individual code blocks. Furthermore, we introduce a novel ensembling method using an effective strategy suggested by MLE-STAR. Our experimental results show that MLE-STAR achieves medals in 64% of the Kaggle competitions on the MLE-bench Lite, significantly outperforming the best alternative.

1. Introduction

The proliferation of machine learning (ML) has driven high-performance applications across diverse real-world scenarios, from fundamental tasks like tabular classification (Chen and Guestrin, 2016; Hollmann et al., 2025; Prokhorenkova et al., 2018) to complex ones such as image denoising (Fan et al., 2019). Despite these advances, developing such models remains a labor-intensive process for data scientists, involving extensive iterative experimentation and data engineering (Hollmann et al., 2023; Nam et al., 2024). To streamline such intensive workflows, recent research has focused on employing large language models (LLMs) (Brown et al., 2020; Team et al., 2024; Touvron et al., 2023) as *machine learning engineering (MLE) agents* (Guo et al., 2024; Hong et al., 2024; Jiang et al., 2025). By harnessing the coding and reasoning capabilities inherent in LLMs (Jain et al., 2025; Jimenez et al., 2024), these agents conceptualize ML tasks as code optimization problems. They then navigate the potential code solutions ultimately producing executable code (*e.g.*, a Python script) based on a provided task description and dataset (see Figure 1).

Despite their promise as pioneering efforts, current MLE agents face several obstacles that limit their effectiveness. First, due to their strong reliance on inherent LLM knowledge, they are often biased toward familiar and frequently used methods (*e.g.*, the scikit-learn library (Pedregosa et al., 2011) for tabular data), neglecting potentially promising task-specific methods. Additionally, these agents (Guo et al., 2024; Jiang et al., 2025) typically employ an exploration strategy that modifies the entire code structure at once in each iteration. This often results in agents pivoting prematurely to other steps (*e.g.*, model selection or hyperparameter tuning) because they lack the ability to perform deep, iterative exploration within specific pipeline components, such as experimenting different feature engineering options extensively.

Contributions. We propose **MLE-STAR**, a novel **ML** **E**ngineering agent that integrates web **S**earch and **TA**rgeted code block **R**efinement (see Figure 2 for an overview). Specifically, generating initial solution code, MLE-STAR utilizes Google Search to retrieve relevant and potentially state-of-the-art

Corresponding author(s): jaehyun.nam@kaist.ac.kr, jinsungyoon@google.com
* This work was done while Jaehyun was a student researcher at Google Cloud.

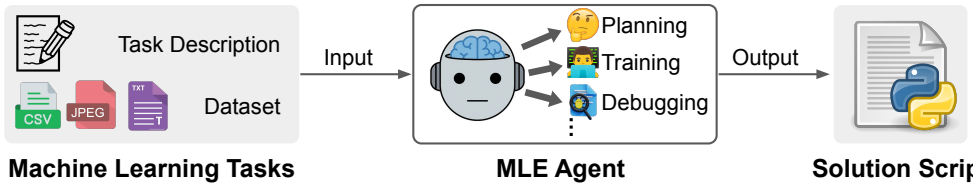

Figure 1 | **Problem setup.** ML Engineering agents are designed to process a task description and datasets across various modalities (*e.g.*, tabular, text, image, audio, etc.) with the objective of determining the optimal solution for a given machine learning problem, such as classification, regression, sequence-to-sequence generation, image denoising, text normalization, etc.

approaches that could be effective towards building a model. Moreover, to improve the solution, MLE-STAR extracts a specific code block that represents a distinct ML pipeline component, such as feature engineering or ensemble building, and then concentrates on exploring strategies that are targeted to that component, using previous attempts as feedback to reflect on. Here, to identify the code block that has the greatest impact on performance, MLE-STAR performs an ablation study that evaluates the contribution of each ML component. This refinement process is repeated, modifying various code blocks (*i.e.*, other ML components). In addition, we introduce a novel method to generate ensembles. MLE-STAR first proposes multiple candidate solutions. Then, instead of relying on a simple voting based on validation scores, MLE-STAR merges these candidates into a single improved solution using an ensemble strategy proposed by the agent itself. This ensemble strategy is iteratively refined based on the performance of the previous strategies.

To verify the effectiveness, we conduct comprehensive evaluations of MLE-STAR using the MLE-bench's Kaggle competitions (Chan et al., 2025). The experimental results demonstrate that MLE-STAR, requiring only minimal human effort (*e.g.*, defining initial prompts that are generalizable to any tasks), significantly outperforms previous methods (Jiang et al., 2025), including those requiring manual labor to collect strategies from Kaggle (Guo et al., 2024). In particular, MLE-STAR achieves a substantial gain in medal achievement, improving it from 25.8% to 63.6% when compared to the top-performing baseline. Additionally, we show that our proposed ensemble technique provides a meaningful improvement to MLE-STAR.

2. Related work

LLM agents. Recent advances in LLMs have led to an active research in autonomous agents. General-purpose agents like ReAct (Yao et al., 2023) and HuggingGPT (Shen et al., 2023) typically use external tools to analyze various problems. Specialized agents, such as Voyager (Wang et al., 2023) for Minecraft or AlphaCode (Li et al., 2022) for code generation, excel in specific domains, often using execution feedback to iteratively improve their approach. Extending these, we introduce MLE-STAR, an LLM agent that specialized in ML tasks.

Automated machine learning. Automated machine learning (AutoML) aims to reduce reliance on human experts by automating end-to-end ML pipelines (Feurer et al., 2022; Jin et al., 2019; LeDell and Poirier, 2020). Auto-WEKA (Kotthoff et al., 2017), TPOT (Olson and Moore, 2016), and recent advances such as AutoGluon (Erickson et al., 2020), have made progress through exploring within predefined model or hyperparameter spaces. AutoML research also specializes in areas such as neural network design (Elsken et al., 2019; Pham et al., 2018; Real et al., 2019; Zoph and Le, 2017), and feature engineering (Fan et al., 2010; Horn et al., 2019; Kanter and Veeramachaneni, 2015; Li et al., 2023; Zhang et al., 2023). However, these methods rely on predefined search spaces, which often

require domain expertise to define. To address this, LLM-based MLE agents (Guo et al., 2024; Jiang et al., 2025), including MLE-STAR, are emerging, since they employ effective exploration strategies directly in the code space, without the need of manually-curated search spaces.

MLE agents. Leveraging coding and reasoning capabilities of LLMs (Jain et al., 2025; Jimenez et al., 2024), research has been conducted on use of LLMs as MLE agents (Hong et al., 2024; Li et al., 2024; Schmidgall et al., 2025), which generate solution code, to automate ML workflows. While MLAB (Huang et al., 2024a) and OpenHands (Wang et al., 2024) take general actions by calling tools to perform ML tasks, several studies specialize in ML automation. AIDE (Jiang et al., 2025) generates candidate solutions in a tree structure to facilitate code space exploration. However, its heavy reliance on the LLM's internal knowledge can lead to outdated or overly simple model choices, and its refinement may prematurely shift focus between pipeline stages. DS-Agent (Guo et al., 2024) uses case-based reasoning (Kolodner, 1992; Watson and Marir, 1994) to discover strategies for solution generation by utilizing manually curated cases (primarily from Kaggle). However, DS-Agent suffers from scalability issues due to its reliance on a manually built case bank, which requires significant human effort and can lead to solutions that are overfit to the source patterns. Also, it restricts applicability to novel task types (like complex multi-modal problems). Our method addresses these limitations. Instead of attempting to explore the broader code space or relying on a static case bank, MLE-STAR strategically explores implementation options for specific ML pipeline components. It also improves scalability by using LLMs with search as tool to retrieve effective models that fit the task beyond the constraints of a fixed case bank.

3. MLE-STAR

We introduce the proposed framework for MLE agents, MLE-STAR, that effectively leverages the coding and reasoning capabilities of LLMs to solve ML tasks. In a nutshell, our approach is based on first generating an initial solution by using web search as a tool (Section 3.1), and then refining solutions via nested loops. The outer loop targets one code block, which corresponds to the specific ML component extracted through an ablation study. The inner loop iteratively refines *only* this block until the outer loop moves to the next target (Section 3.2). We propose a novel ensemble method that improves the performance using the plan proposed by LLMs, which is iteratively refined (Section 3.3). To mitigate potential undesirable behaviors from LLMs, such as using test sample statistics for missing value imputation, we introduce specific modules (detailed in Section 3.4). The prompts and algorithms used in each step can be found in Appendix A and B, respectively.

Problem setup. Formally, our goal is to find an optimal solution $s^* = \arg\max_{s \in \mathcal{S}} h(s)$, where $\mathcal{S}$ is the space of possible solutions (*i.e.*, Python scripts) and $h : \mathcal{S} \to \mathbb{R}$ is a score function (*e.g.*, validation accuracy) (Jiang et al., 2025). To obtain s^*, we propose a multi-agent framework $\mathcal{A}$, which takes datasets $\mathcal{D}$ (that might contain multiple files) and a task description $\mathcal{T}_{\texttt{task}}$ (which includes task types, data modalities, score functions, etc.) as input.[1] Here, $\mathcal{A}$ consists of n LLM agents $(\mathcal{A}_1, \cdots, \mathcal{A}_n)$. Each agent $\mathcal{A}_i$ possesses specific functionalities, which are elaborated upon in following sections.

3.1. Generating an initial solution using web search as a tool

Candidate model search. MLE-STAR starts by generating an initial solution. For high performance in ML tasks, selecting the appropriate model is paramount. However, relying solely on an LLM for model suggestions can lead to suboptimal choices. For instance, we observe that LLMs propose models

[1] MLE-STAR works across any data modalities (*e.g.*, tabular, image, text, audio) and task types (*e.g.*, classification, image-to-image, sequence-to-sequence) – it is not restricted to specific inputs or objectives.

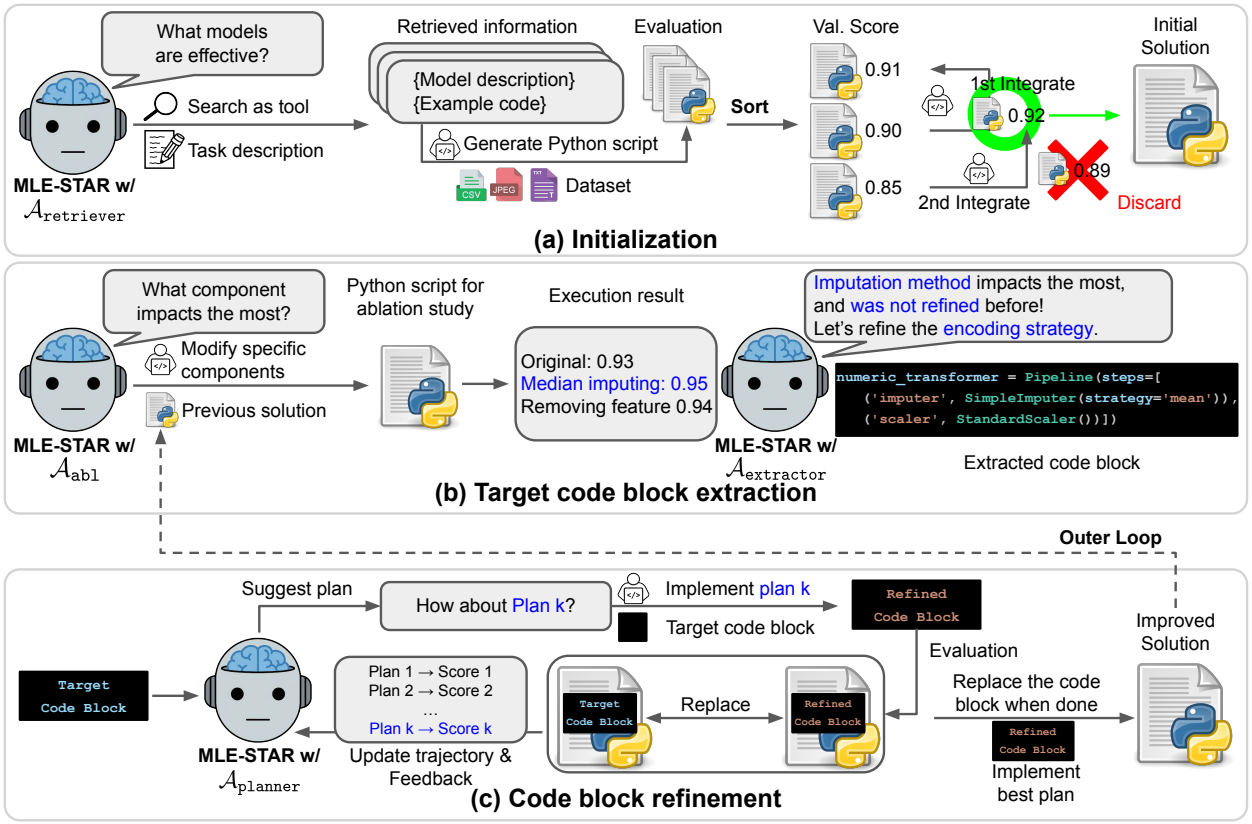

Figure 2 | **Overview of MLE-STAR.** (a) Using search as a tool, MLE-STAR retrieves task-specific models and uses them to generate an initial solution. (b) In each refinement step, MLE-STAR performs an ablation study to extract the code block that have the greatest impact. Previously modified code blocks are also provided as feedback for diversity. (c) The extracted code block is iteratively refined based on plans suggested by the LLM, which explores various plans using previous experiments as feedback (*i.e.*, inner loop), and the target code block is also selected repeatedly (*i.e.*, outer loop, where the improved solution of (c) becomes the previous solution in (b)).

like logistic regression (Pedregosa et al., 2011) even for competitions like jigsaw-toxic-comment-classification, which is a text classification task, potentially because LLMs favor familiar patterns from their pre-training data over up-to-date information. To mitigate this, we propose using web search as a tool for MLE-STAR first to retrieve M effective, state-of-the-art models for the given task. This retrieved context is then used to guide the LLM in generating a more informed initial solution. Formally:

$$\{\mathcal{T}^i_{\texttt{model}}, \mathcal{T}^i_{\texttt{code}}\}_{i=1}^{M} = \mathcal{A}_{\texttt{retriever}}(\mathcal{T}_{\texttt{task}}), \tag{1}$$

where $\mathcal{T}_{\texttt{model}}$ represents the description of a retrieved model, while $\mathcal{T}_{\texttt{code}}$ provides corresponding example code. This example code is needed since the LLM can be unfamiliar with the model and cannot generate the executable code without proper guidance. Then, MLE-STAR involves evaluating of the performance of model i. To achieve this, candidate evaluation agent $\mathcal{A}_{\texttt{init}}$ first generates code, $s^i_{\texttt{init}}$, using the retrieved model to solve the given ML task. This process is formally defined as:

$$s^i_{\texttt{init}} = \mathcal{A}_{\texttt{init}}(\mathcal{T}_{\texttt{task}}, \mathcal{T}^i_{\texttt{model}}, \mathcal{T}^i_{\texttt{code}}). \tag{2}$$

We evaluate the performance of each s using a task-specific metric h on dataset $\mathcal{D}$. We denote the resulting score by $h(s)$, which encapsulates the entire process done in s: splitting $\mathcal{D}$ into training and validation sets, training the model specified in s using the training data, and calculating h on

the validation data. The performance for $s^i_{\texttt{init}}$ is thus $h(s^i_{\texttt{init}})$. As a result, a set of code scripts $\mathcal{S}_{\texttt{init}} = \{s^1_{\texttt{init}}, \cdots, s^M_{\texttt{init}}\}$ and their performance scores $\{h(s^1_{\texttt{init}}), \cdots, h(s^M_{\texttt{init}})\}$ are obtained.

Merging candidate models for initial solution. After the evaluation of the M retrieved models, a consolidated initial solution s_0 is constructed through an iterative merging procedure. Specifically, we first define π be a permutation of the indices such that the scores are sorted in descending order: $h(s^{\pi(1)}_{\texttt{init}}) \geq h(s^{\pi(2)}_{\texttt{init}}) \geq \cdots \geq h(s^{\pi(M)}_{\texttt{init}})$. Then, we initialize the initial solution s_0 with the top-performing script, and record the current best score, *i.e.*, $s_0 \leftarrow s_{(1)}$, $h_{\texttt{best}} \leftarrow h(s_0)$, where $s_{(k)}$ denote the script $s^{\pi(k)}_{\texttt{init}}$ for simplicity. Finally, we sequentially attempt to incorporate the remaining scripts $s_{(k)}$ for $k = 2, \cdots, M$ into s_0. For each k, MLE-STAR creates a candidate merged script by leveraging an agent $\mathcal{A}_{\texttt{merger}}$ that attempts to integrate $s_{(k)}$ into the current s_0. Formally,

$$s_0 \leftarrow \mathcal{A}_{\texttt{merger}}(s_0, s_{(k)}), \; h_{\texttt{best}} \leftarrow h(s_0) \tag{3}$$

where, $\mathcal{A}_{\texttt{merger}}$ is guided to introduce a simple average ensemble to merge multiple models. Finally, we merge the models until the validation score $h_{\texttt{best}}$ no longer improves (see Appendix B).

3.2. Refining a code block for solution improvement

The iterative refinement phase begins with an initial solution s_0 and proceeds for a predetermined number of T outer loop steps, indexed by $t = 0, 1, \cdots, T-1$. At each step t, the goal is to improve the current solution s_t to obtain s_{t+1}, optimizing for a performance metric h. This process involves two main stages: targeted code block extraction and code block refinement.

Targeted code block extraction. To effectively explore specialized improvement strategies, MLE-STAR identifies and targets specific code blocks within the ML pipeline represented by s_t. This selection is guided by an ablation study performed by an agent $\mathcal{A}_{\texttt{abl}}$. Specifically, the agent $\mathcal{A}_{\texttt{abl}}$ generates a code a_t designed to perform an ablation study on s_t. This script creates variations of s_t by modifying or disabling specific components. To encourage exploration of different pipeline parts across iterations, $\mathcal{A}_{\texttt{abl}}$ receives the summaries of previous ablation studies $\{\mathcal{T}^i_{\texttt{abl}}\}_{i=0}^{t-1}$ as input:

$$a_t = \mathcal{A}_{\texttt{abl}}(s_t, \{\mathcal{T}^i_{\texttt{abl}}\}_{i=0}^{t-1}). \tag{4}$$

Then, a_t is executed, producing output results r_t (see examples in Appendix C). A summarization module $\mathcal{A}_{\texttt{summarize}}$ processes the script and its results to generate a concise ablation summary $\mathcal{T}^t_{\texttt{abl}}$:

$$\mathcal{T}^t_{\texttt{abl}} = \mathcal{A}_{\texttt{summarize}}(a_t, r_t). \tag{5}$$

Finally, MLE-STAR selects a critical block which has the most significant impact on performance. To achieve this, an extractor module $\mathcal{A}_{\texttt{extractor}}$ analyzes the $\mathcal{T}^t_{\texttt{abl}}$ to identify the code block c_t within s_t whose modification had the most significant impact on performance. To prioritize refinement of blocks not previously targeted, the set of already refined blocks $\{c_i\}_{i=0}^{t-1}$ is provided as context:

$$c_t, p_0 = \mathcal{A}_{\texttt{extractor}}(\mathcal{T}^t_{\texttt{abl}}, s_t, \{c_i\}_{i=0}^{t-1}). \tag{6}$$

Here, MLE-STAR also generates the initial plan p_0 for code block refinement at the same time, since $\mathcal{T}_{\texttt{abl}}$ can provide a good starting point by analyzing the modification of corresponding component.

Code block refinement. Once the targeted code block c_t is defined, MLE-STAR explores various refinement strategies to improve the metric h. This involves an inner loop exploring K potential refinement for c_t. An agent $\mathcal{A}_{\texttt{coder}}$ first implements p_0, transforming c_t into a refined block c^0_t, *i.e.*, $c^0_t = \mathcal{A}_{\texttt{coder}}(c_t, p_0)$. A candidate solution s^0_t is formed by substituting c^0_t into s_t:

$$s^0_t = s^t.\texttt{replace}(c_t, c^0_t), \tag{7}$$

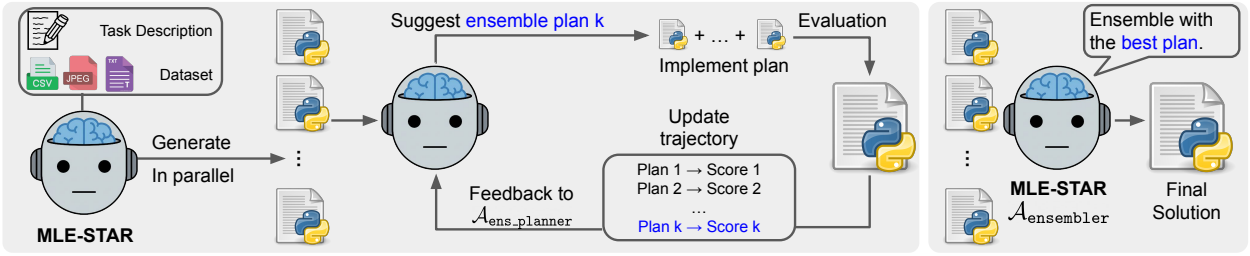

Figure 3 | **Ensembling solutions.** MLE-STAR iteratively proposes effective ensemble strategies based on previous attempts, integrating multiple solutions generated in parallel into a single solution.

where, `replace` denotes the code replacement operation. Finally, the performance $h(s_t^0)$ is evaluated.

To discover potentially more effective or novel refinement strategies, MLE-STAR iteratively generates and evaluates further plans. For $k = 1, \cdots, K-1$, a planning agent $\mathcal{A}_{\texttt{planner}}$ proposes the next plan p_k. This agent leverages the previous attempts within the current outer step t as feedback:

$$p_k = \mathcal{A}_{\texttt{planner}}(c_t, \{(p_j, h(s_t^j))\}_{j=0}^{k-1}). \tag{8}$$

For each plan p_k, the coding agent generates the corresponding refined block, *i.e.*, $c_t^k = \mathcal{A}_{\texttt{coder}}(c_t, p_k)$, creates the candidate solution $s_t^k = s_t.\texttt{replace}(c_t, c_t^k)$, and evaluates its performance $h(s_t^k)$. After exploring K refinement strategies (indexed $k = 0, \cdots, K-1$), the best-performing candidate solution is identified: $k^* = \arg\max_{k \in \{0,\cdots,K-1\}} h(s_t^k)$. The solution for the next outer step, s_{t+1}, is updated to $s_t^{k^*}$ only if an improvement over s_t is found. This iterative process continues until $t = T$.

3.3. Further improvement by exploring ensemble strategies

To further improve upon the best single solution generated, we introduce a novel ensembling procedure (Figure 3). Standard practice might involve generating multiple candidate solutions and selecting the one with the highest score (Ichihara et al., 2025) according to metric h. However, analogous to model ensembling, we posit that suboptimal solutions might contain complementary strengths, and combining multiple solutions could lead to superior performance compared to relying on any single one. Therefore, we employ the planning capabilities of MLE-STAR to automatically discover effective strategies for ensembling. Specifically, let $\{s_l\}_{l=1}^L$ be a set of L distinct solutions obtained (*e.g.*, from parallel runs of the process described earlier). Our goal is to find an effective ensemble plan e that merges these solutions, which mirrors the structure of the targeted code block refinement stage. We start with an initial ensemble plan e_0 (*e.g.*, a simple strategy like averaging the final predictions obtained from the models trained using each solution s_l), proposed by MLE-STAR itself. After the performance $h(s_{\text{ens}}^0)$ for the initial plan e_0 is calculated, for a fixed number of iterations, $r = 1, \cdots, R$, the planning agent $\mathcal{A}_{\texttt{ens\_planner}}$, specialized in suggesting ensemble plans, proposes subsequent ensemble plans e_r. This agent uses the history of previously attempted ensemble plans and their resulting performance as feedback, *i.e.*, $e_r = \mathcal{A}_{\texttt{ens\_planner}}(\{s_l\}_{l=1}^L, \{(e_j, h(s_{\text{ens}}^j))\}_{j=0}^{r-1})$. Each e_r is implemented via $\mathcal{A}_{\texttt{ensembler}}$ to obtain s_{ens}^r:

$$s_{\text{ens}}^r = \mathcal{A}_{\texttt{ensembler}}(e_r, \{s_l\}_{l=1}^L). \tag{9}$$

Finally, after exploring R ensemble strategies, the ensemble result that achieves the highest performance is selected as the final output, yielding the final ensembled result $s_{\text{ens}}^* = s_{\text{ens}}^{r^*}$: $r^* = \arg\max_{r \in \{0,\ldots,R\}} h(s_{\text{ens}}^r)$. This procedure allows MLE-STAR to autonomously explore and identify potentially novel and effective ways to combine multiple complex solutions.

3.4. Additional modules for robust MLE agents

Debugging agent. We detail the design of our debugging agent within MLE-STAR. If the execution of a Python script s triggers an error, resulting in a record $\mathcal{T}_{\texttt{bug}}$ (*e.g.*, a traceback), MLE-STAR employs a debugging module $\mathcal{A}_{\texttt{debugger}}$ to attempt correction. This process iteratively updates the script:

$$s \leftarrow \mathcal{A}_{\texttt{debugger}}(s, \mathcal{T}_{\texttt{bug}}). \tag{10}$$

The debugging step is repeated until either the script executes successfully, or a predefined maximum number of debugging rounds is reached. If the bug cannot be resolved, MLE-STAR proceeds to the next task using the latest version of the script that is known to be executable.

Data leakage checker. We observe that LLM-generated Python scripts might have the risk of introducing data leakage, for example, by improperly accessing information from a test dataset during training dataset preparation (see Figure 6). To address this, we introduce a checker agent, $\mathcal{A}_{\texttt{leakage}}$, which analyzes the solution script s prior to its execution. Recognizing that full-script analysis can be inefficient for lengthy code, we adopt a targeted approach. First, we extract the code block $c_{\texttt{data}}$ where data preprocessing is done. Second, $c_{\texttt{data}}$ is passed to the checker. If $\mathcal{A}_{\texttt{leakage}}$ detects potential data leakage, it generates a corrected version $c^*_{\texttt{data}}$: $c^*_{\texttt{data}} = \mathcal{A}_{\texttt{leakage}}(c_{\texttt{data}})$. Finally, the original script s is updated by replacing the identified segment with its corrected version: $s \leftarrow s.\texttt{replace}(c_{\texttt{data}}, c^*_{\texttt{data}})$. If no leakage is detected in $c_{\texttt{data}}$ by $\mathcal{A}_{\texttt{leakage}}$, the script s remains unmodified. All generated solutions are passed through a data leakage checker, $\mathcal{A}_{\texttt{leakage}}$, prior to their execution for evaluation.

Data usage checker. We observe that LLM-generated scripts sometimes neglect using provided data sources, focusing solely on simple formats like CSVs (see Figure 7). To ensure the utilization of all relevant provided data, MLE-STAR introduces a data usage checker agent, $\mathcal{A}_{\texttt{data}}$. Specifically, before MLE-STAR starts refinement, $\mathcal{A}_{\texttt{data}}$ checks the initial solution s_0 along with the task description $\mathcal{T}_{\texttt{task}}$. If relevant provided data is not adequately used, $\mathcal{A}_{\texttt{data}}$ revises the initial script as:

$$s_0 \leftarrow \mathcal{A}_{\texttt{data}}(s_0, \mathcal{T}_{\texttt{task}}). \tag{11}$$

4. Experiments

In this section, we validate the effectiveness of MLE-STAR using 22 Kaggle competitions from MLE-bench Lite (Chan et al., 2025). Our results demonstrate that MLE-STAR significantly outperforms baselines, including those employing various LLMs (Section 4.1). Furthermore, we show that using better models and leveraging our proposed ensemble strategy effectively improves performance (Section 4.2). We also provide the example solutions generated by MLE-STAR, in Appendix D.

Common setup. All experiments are conducted on 22 Kaggle competitions from MLE-bench Lite (Chan et al., 2025) using three random seeds and Gemini-2.0-Flash, unless otherwise specified. Here, we use an agent $\mathcal{A}_{\texttt{test}}$, which takes the task description and the final solution as input, and outputs the code that incorporates loading test sample and creating a submission file (see Appendix E for details). MLE-STAR begins by retrieving four model candidates. MLE-STAR refines for four inner loops, while exploring four outer loops. For ensemble, MLE-STAR generates two solutions in parallel, and explore ensemble strategies for five rounds. Following the MLE-bench's setup, we set a maximum time limit of 24 hours for a fair comparison (see computation analysis in Appendix F). We primarily consider AIDE (Jiang et al., 2025) as our main baseline, given its state-of-the-art performance on MLE-bench. It is important to note that other baselines often limit their generalizability across various task types (*e.g.*, audio classification, sequence-to-sequence), frequently showcasing results only on simpler modalities like tabular (Hong et al., 2024; Li et al., 2024). For instance, DS-Agent (Guo et al., 2024) requires a manually constructed case bank, and their current GitHub repository lacks cases for audio classification, sequence-to-sequence, image classification, etc.

Table 1 | **Main results from MLE-bench Lite.** Each experiment is repeated using three seeds, except for o1-preview (AIDE) and GPT-4o (AIDE), which use 16 and 36 seeds, respectively. All results are taken from the GitHub repository of MLE-bench paper (Chan et al., 2025), except for the model using Gemini. Scores represent the mean and one standard error of the mean.

Model	Made Submission (%)	Valid Submission (%)	Above Median (%)	Bronze (%)	Silver (%)	Gold (%)	Any Medal (%)
MLE-STAR (Ours)							
gemini-2.5-pro	**100.0**$_{\pm 0.0}$	**100.0**$_{\pm 0.0}$	**83.3**$_{\pm 4.6}$	6.1$_{\pm 3.0}$	**21.2**$_{\pm 5.1}$	**36.4**$_{\pm 6.0}$	**63.6**$_{\pm 6.0}$
gemini-2.0-flash	95.5$_{\pm 2.6}$	95.5$_{\pm 2.6}$	63.6$_{\pm 6.0}$	**9.1**$_{\pm 3.6}$	4.5$_{\pm 2.6}$	30.3$_{\pm 5.7}$	43.9$_{\pm 6.2}$
AIDE (Jiang et al., 2025)							
gemini-2.0-flash	87.9$_{\pm 4.0}$	78.8$_{\pm 5.0}$	39.4$_{\pm 6.0}$	4.5$_{\pm 2.6}$	9.1$_{\pm 3.5}$	12.1$_{\pm 4.0}$	25.8$_{\pm 5.4}$
o1-preview	99.7$_{\pm 0.3}$	90.3$_{\pm 1.6}$	58.2$_{\pm 2.6}$	4.8$_{\pm 1.1}$	11.1$_{\pm 1.7}$	20.7$_{\pm 2.2}$	36.6$_{\pm 2.6}$
gpt-4o	82.1$_{\pm 1.4}$	65.7$_{\pm 1.7}$	29.9$_{\pm 1.6}$	3.4$_{\pm 0.6}$	5.8$_{\pm 0.8}$	9.3$_{\pm 1.0}$	18.6$_{\pm 1.4}$
llama-3.1-405b-instruct	72.7$_{\pm 5.5}$	51.5$_{\pm 6.2}$	18.2$_{\pm 4.7}$	0.0$_{\pm 0.0}$	4.5$_{\pm 2.6}$	6.1$_{\pm 2.9}$	10.6$_{\pm 3.8}$
claude-3-5-sonnet	81.8$_{\pm 4.7}$	66.7$_{\pm 5.8}$	33.3$_{\pm 5.8}$	3.0$_{\pm 2.1}$	6.1$_{\pm 2.9}$	10.6$_{\pm 3.8}$	19.7$_{\pm 4.9}$
MLAB (Huang et al., 2024a)							
gpt-4o	84.8$_{\pm 4.4}$	63.6$_{\pm 5.9}$	7.6$_{\pm 3.3}$	3.0$_{\pm 2.1}$	1.5$_{\pm 1.5}$	1.5$_{\pm 1.5}$	6.1$_{\pm 2.9}$
OpenHands (Wang et al., 2024)							
gpt-4o	81.8$_{\pm 4.7}$	71.2$_{\pm 5.6}$	16.7$_{\pm 4.6}$	3.0$_{\pm 2.1}$	3.0$_{\pm 2.1}$	6.1$_{\pm 2.9}$	12.1$_{\pm 4.0}$

Table 2 | Comparison with DS-Agent.

Task	Metric	DS-Agent	**MLE-STAR**
WBY	MAE (↓)	213	**166**
MCC	RMLSE (↓)	0.2964	**0.2911**
ST	Accuracy (↑)	0.7982	**0.8091**
ES	AUROC (↑)	0.8727	**0.9101**

Table 3 | Performance with Claude-Sonnet-4.

Task	Metric	Gemini-2.0-flash	**Sonnet 4**
DDD	RMSE (↓)	0.0681	**0.0155**
DBI	Log Loss (↓)	0.4535	**0.3114**
SAI	Log Loss (↓)	0.2797	**0.2610**
WCR	AUROC (↑)	**0.9903**	0.9888

4.1. Main results

Quantitative results. As demonstrated in Table 1, MLE-STAR significantly enhances the performance of various baseline models. For instance, when applied to Gemini-2.0-Flash, MLE-STAR improves AIDE's any medal achieving rates in Kaggle competitions from 25.8% to 43.9%, representing an improvement of over 18 percentage points, and rate of above median from 39.4% to 63.6%. Notably, MLE-STAR with Gemini-2.0-Flash also substantially outperforms AIDE using a powerful reasoning model (*i.e.*, o1-preview) in terms of achieving gold medals in 10% more tasks. Moreoever, using Gemini-2.5-Pro, MLE-STAR shows a medal achievement rate of over 60%.

Comparison to DS-Agent. While DS-Agent (Guo et al., 2024) shows competitive results on ML tasks, it necessitates human effort to curate its case bank from Kaggle. Consequently, a direct comparison between DS-Agent and AIDE or our method is not feasible, as collecting tasks across diverse modalities, such as audio classification or image denoising, requires additional effort. Nevertheless, we utilize four tabular classification tasks, *i.e.*, wild-blueberry-yield (WBY), media-campaign-cost (MCC), spaceship-titanic (ST), and enzyme-substrate (ES), the same ones employed during DS-Agent's development stage (Guo et al., 2024), for a comparison. All experiments are done for 5 seeds following the original setup. As shown in Table 2, MLE-STAR using Gemini-2.0-Flash significantly outperforms DS-Agent even without human efforts. See Appendix G for additional results, including comparison with AutoGluon (Erickson et al., 2020).

Table 4 | **Ablation on ensemble strategy.** Experiment results on MLE-bench Lite, repeated three seeds using Gemini-2.0-Flash. Scores represent the mean and one standard error of the mean.

Ensemble strategy	Made Submission (%)	Valid Submission (%)	Above Median (%)	Bronze (%)	Silver (%)	Gold (%)	Any Medal (%)
AIDE (Jiang et al., 2025)							
None	87.9±4.0	78.8±5.0	39.4±6.0	4.5±2.6	9.1±3.5	12.1±4.0	25.8±5.4
MLE-STAR (Ours)							
None	**95.5**±2.6	**95.5**±2.6	57.6±6.1	7.6±3.3	4.5±2.6	25.8±5.4	37.9±6.0
Best-of-N	**95.5**±2.6	**95.5**±2.6	62.1±6.0	6.1±3.0	7.6±3.3	28.8±5.6	42.4±6.1
Average ensemble	**95.5**±2.6	**95.5**±2.6	60.6±6.1	6.1±3.0	**12.1**±4.0	25.8±9.4	**43.9**±6.2
Ours	**95.5**±2.6	**95.5**±2.6	**63.6**±6.0	**9.1**±3.6	4.5±2.6	**30.3**±5.7	**43.9**±6.2

4.2. Ablation studies

Performance with reasoning models. First of all, as shown in Table 1, Gemini-2.5-Pro yields better performance than using Gemini-2.0-Flash. For example, in denoising-dirty-documents cometition, MLE-STAR wigh Gemini-2.0-Flash scored above the median across all three seeds, failing to achieve any medals. However, when using Gemini-2.5-Pro, MLE-STAR achieves two gold medals and one silver medal. These results demonstrate that MLE-STAR is designed to harness the advancements of rapidly improving reasoning-based LLMs.

In addition, we conducted additional experiments using Claude-Sonnet-4. As shown in Table 3, other models besides Gemini also show promising results, proving compatibility and generalizability in terms of model types. Here, we select four different type of competitions: image-to-image (denoising-dirty-documents; DDD), image classification (dog-breed-identification; DBI), text classification (spooky-author-identification, SAI), and audio classification (the-icml-2013-whale-challenge-right-whale-redux; WCR). We run each competition for three seeds. These results indicates that our framework is also compatible and generalizable in terms of LLM type.

Effectiveness of proposed ensemble method. As highlighted in Table 4, MLE-STAR demonstrates a significant performance improvement over the competing baseline, *i.e.*, AIDE, achieving over a 12% higher rate of obtaining any medal *even without* additional ensemble strategy. Notably, by ensembling multiple solution candidates, our approach yields even greater performance gains, *i.e.*, MLE-STAR consistently improves the success rate for achieving any medal (and specifically gold medals), also surpassing the median human expert's performance by a larger margin compared to scenarios where this ensembling method is not used. While simpler strategies, such as selecting the solution with the best validation score or averaging final submissions, also offer benefits, MLE-STAR shows stronger effectiveness, *e.g.*, leading to a higher number of gold medals.

5. Discussion

Qualitative observations on selected models. Figure 4 illustrates the model usage of two MLE agents: AIDE and MLE-STAR. AIDE primarily employs ResNet (He et al., 2016) for image classification. However, ResNet, released in 2015, is now considered outdated and can result in suboptimal performance. In contrast, our MLE-STAR primarily utilizes more recent and competitive models like EfficientNet (Tan and Le, 2019) or ViT (Dosovitskiy et al., 2021), leading to the performance gain, winning 37% of the medals, more than AIDE, which wins 26% of the image classification challenges.

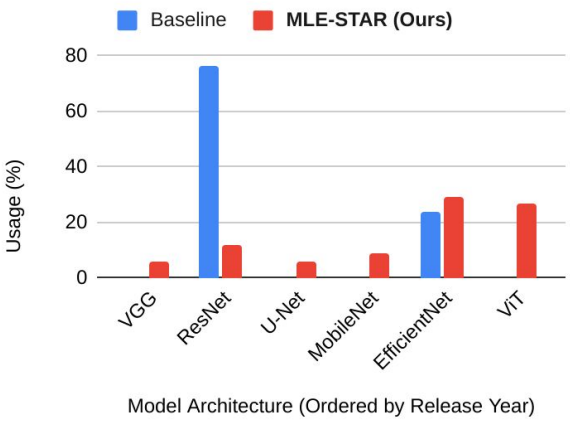

Figure 4 | **Model usage** (%) on image classification competitions. Other models (11.7%), which are used by MLE-STAR, are omitted.

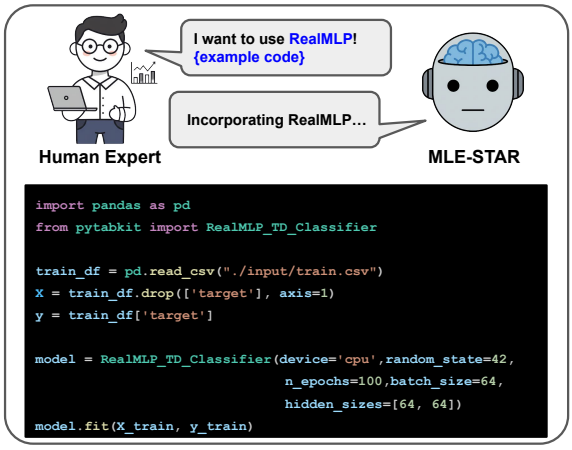

Figure 5 | **Human intervention.** By manually adding a model description, MLE-STAR integrates its training into the framework.

Improperly imputed missing values

```
# Combined DataFrame for Consistent Preprocessing
all_df = pd.concat([train_df, test_df])
# Impute Numerical features with median
num_cols = ['Age', 'RoomService', 'FoodCourt']
for col in num_cols:
  if all_df[col].isnull().any():
      median_val = all_df[col].median()
      all_df[col] = all_df[col].fillna(median_val)
```

Python script after refined by data leakage checker

```
# Preprocess Training Set and get fit statistics
X_train_processed, stats =
      preprocess_data(X_train, fit_stats=None)
# Preprocess Test Set using statistics from Train Set
X_test_processed =
      preprocess_data(X_test, fit_stats=stats)
```

Figure 6 | MLE-STAR's **data leakage checker** introduces appropriate preprocessing.

Python script before refined by data usage checker

```
data = pd.read_csv('./input/train.csv')
```

Python script after refined by data leakage checker

```
def process_xyz_files(df): # Features using XYZ files
  def volume(filepath):
      lines = open(filepath, 'r').readlines()
      a_vec = list(map(float, lines[2].split()[1:]))
      b_vec = list(map(float, lines[3].split()[1:]))
      c_vec = list(map(float, lines[4].split()[1:]))
      volume = np.dot(a_vec, np.cross(b_vec, c_vec))
      return abs(volume)
  for index, row in df.iterrows():
      file_path = os.path.join(row['id'],'geometry.xyz')
      df.loc[index, 'atomic_volume'] = volume(file_path)
  return df
data = process_xyz_files(data) # Process the train data
```

Figure 7 | MLE-STAR's **data usage checker** captures previously unused information.

Human intervention. MLE-STAR readily adopts even more recent models with minimal human intervention. While MLE-STAR automatically constructs a model description $\{\mathcal{T}_{\texttt{model}}, \mathcal{T}_{\texttt{code}}\}$ using search as tool, a natural extension involves leveraging human expertise for this construction. As shown in Figure 5, by manually adding a model description for RealMLP (Holzmüller et al., 2024), MLE-STAR successfully integrates its training into the framework, a model not previously retrieved. In addition, users can also specify the target code blocks by replacing the ablation summary with manually written instructions.

Misbehavior of LLMs and corrections. We observe that while the code generated by the LLM executed correctly, their content is sometime unrealistic, exhibiting hallucination. For example, Figure 6 illustrates an impractical approach where test data is preprocessed using its own statistics. Since test data must remain unseen, correction in the code is necessitated, for which, MLE-STAR employs a data leakage checker $\mathcal{A}_{\texttt{leakage}}$ to identify such issues in the generated Python script. If a problem is detected, MLE-STAR refines the code. As shown in the Figure, MLE-STAR successfully identifies the issue and modifies the code by, first extracting statistics from the training data and then preprocessing the test

Table 5 | Improvement failure when not using data leakage checker $\mathcal{A}_{\texttt{leakage}}$ on spaceship-titanic competition.

Metric	Accuracy (↑)
Validation	0.8188 → **0.8677**
Test	**0.8033** → 0.7343

Table 6 | Ablation study of data usage checker $\mathcal{A}_{\texttt{data}}$ on nomad2018-predicting competition.

Model	$\mathcal{A}_{\texttt{data}}$	RMSLE (↓)
MLE-STAR	✗	0.0591
MLE-STAR	✓	**0.0559**

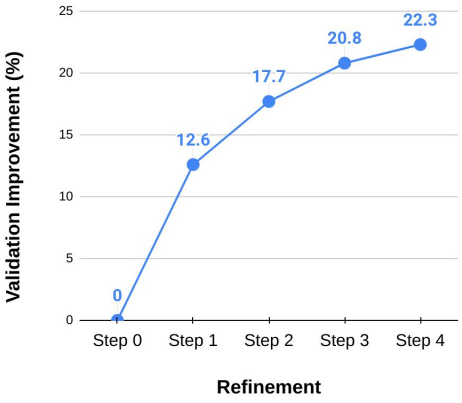

Figure 8 | Solution refinement trajectory.

data using these calculated statistics. In addition, the improvement process can fail to generalize when $\mathcal{A}_{\texttt{leakage}}$ is not employed, as exemplified in Table 5. In this example, the validation accuracy (*i.e.*, the target objective) improves, but the test accuracy drops significantly. This is attributed to the LLM performing feature engineering using the target variable `Transported`, which is not accessible in the test set, leading to data leakage and subsequently, poor test performance.

We also observe that LLMs often generate Python scripts that overlook some of the provided data sources. For example, in the nomad2018-predicting competition, Gemini-2.0-Flash solely loads train.csv, neglecting the use of geometry.xyz (see Figure 7). To address this, MLE-STAR employs $\mathcal{A}_{\texttt{data}}$, which reexamines the task description to ensure that all given data is utilized. As shown in Figure 7, this design enables MLE-STAR to incorporate previously neglected data. As a result, performance is significantly improved, as shown in Table 6.

Progressive improvement via MLE-STAR refinement. This section details the progressive improvement of solutions achieved by MLE-STAR, as measured by validation metrics. Given the task-specific nature of evaluation metrics, we report the average relative error reduction (%) across the all 22 challenges in MLE-bench Lite (Chan et al., 2025). This metric measures the extent to which MLE-STAR reduces the error of an initial solution. Figure 8 demonstrates a consistent improvement as MLE-STAR proceeds through its refinement steps, which each step focusing on refining a single code block via an inner loop. Significantly, the magnitude of improvement is notable in the early refinement stages. We posit that this stems from MLE-STAR's ablation study module which helps to target the most influential code blocks for modification first.

6. Conclusion

We propose MLE-STAR, a novel MLE agent designed for various ML tasks. Our key idea is to utilize a search engine to retrieve effective models and then explore various strategies targeting specific ML pipeline components to improve the solution. The effectiveness of MLE-STAR is validated by winning medals in 64% (where 36% are gold medals) of the MLE-bench Lite Kaggle competitions.

Limitation. Since Kaggle competitions are publicly accessible, there is a potential risk that LLMs might have been trained with the relevant discussions about the challenge. Nevertheless, we show that MLE-STAR's solution is sufficiently novel (using LLM as a judge) compared to the discussions on Kaggle (see Appendix H).

References

T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 2020.

J. S. Chan, N. Chowdhury, O. Jaffe, J. Aung, D. Sherburn, E. Mays, G. Starace, K. Liu, L. Maksin, T. Patwardhan, et al. Mle-bench: Evaluating machine learning agents on machine learning engineering. *International Conference on Learning Representations*, 2025.

T. Chen and C. Guestrin. Xgboost: A scalable tree boosting system. *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016.

A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. *International Conference on Learning Representations*, 2021.

T. Elsken, J. H. Metzen, and F. Hutter. Neural architecture search: A survey. *Journal of Machine Learning Research*, 2019.

N. Erickson, J. Mueller, A. Shirkov, H. Zhang, P. Larroy, M. Li, and A. Smola. Autogluon-tabular: Robust and accurate automl for structured data. *arXiv preprint arXiv:2003.06505*, 2020.

L. Fan, F. Zhang, H. Fan, and C. Zhang. Brief review of image denoising techniques. *Visual computing for industry, biomedicine, and art*, 2019.

W. Fan, E. Zhong, J. Peng, O. Verscheure, K. Zhang, J. Ren, R. Yan, and Q. Yang. Generalized and heuristic-free feature construction for improved accuracy. *SIAM International Conference on Data Mining*, 2010.

M. Feurer, K. Eggensperger, S. Falkner, M. Lindauer, and F. Hutter. Auto-sklearn 2.0: Hands-free automl via meta-learning. *Journal of Machine Learning Research*, 2022.

S. Guo, C. Deng, Y. Wen, H. Chen, Y. Chang, and J. Wang. DS-agent: Automated data science by empowering large language models with case-based reasoning. *International Conference on Machine Learning*, 2024.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

N. Hollmann, S. Müller, and F. Hutter. Large language models for automated data science: Introducing caafe for context-aware automated feature engineering. *Advances in Neural Information Processing Systems*, 2023.

N. Hollmann, S. Müller, L. Purucker, A. Krishnakumar, M. Körfer, S. B. Hoo, R. T. Schirrmeister, and F. Hutter. Accurate predictions on small data with a tabular foundation model. *Nature*, 2025.

D. Holzmüller, L. Grinsztajn, and I. Steinwart. Better by default: Strong pre-tuned mlps and boosted trees on tabular data. *Advances in Neural Information Processing Systems*, 2024.

S. Hong, Y. Lin, B. Liu, B. Liu, B. Wu, C. Zhang, C. Wei, D. Li, J. Chen, J. Zhang, et al. Data interpreter: An llm agent for data science. *arXiv preprint arXiv:2402.18679*, 2024.

F. Horn, R. Pack, and M. Rieger. The autofeat python library for automated feature engineering and selection. *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, 2019.

X. Hu, Z. Zhao, S. Wei, Z. Chai, Q. Ma, G. Wang, X. Wang, J. Su, J. Xu, M. Zhu, et al. Infiagent-dabench: Evaluating agents on data analysis tasks. *arXiv preprint arXiv:2401.05507*, 2024.

Q. Huang, J. Vora, P. Liang, and J. Leskovec. Mlagentbench: Evaluating language agents on machine learning experimentation. *International Conference on Machine Learning*, 2024a.

Y. Huang, J. Luo, Y. Yu, Y. Zhang, F. Lei, Y. Wei, S. He, L. Huang, X. Liu, J. Zhao, et al. Da-code: Agent data science code generation benchmark for large language models. *arXiv preprint arXiv:2410.07331*, 2024b.

Y. Ichihara, Y. Jinnai, T. Morimura, K. Abe, K. Ariu, M. Sakamoto, and E. Uchibe. Evaluation of best-of-n sampling strategies for language model alignment. *Transactions on Machine Learning Research*, 2025.

N. Jain, K. Han, A. Gu, W.-D. Li, F. Yan, T. Zhang, S. Wang, A. Solar-Lezama, K. Sen, and I. Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *International Conference on Learning Representations*, 2025.

Z. Jiang, D. Schmidt, D. Srikanth, D. Xu, I. Kaplan, D. Jacenko, and Y. Wu. Aide: Ai-driven exploration in the space of code. *arXiv preprint arXiv:2502.13138*, 2025.

C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan. Swe-bench: Can language models resolve real-world github issues? *International Conference on Learning Representations*, 2024.

H. Jin, Q. Song, and X. Hu. Auto-keras: An efficient neural architecture search system. *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.

L. Jing, Z. Huang, X. Wang, W. Yao, W. Yu, K. Ma, H. Zhang, X. Du, and D. Yu. Dsbench: How far are data science agents to becoming data science experts? *International Conference on Learning Representations*, 2025.

J. M. Kanter and K. Veeramachaneni. Deep feature synthesis: Towards automating data science endeavors. *IEEE International Conference on Data Science and Advanced Analytics*, 2015.

J. L. Kolodner. An introduction to case-based reasoning. *Artificial intelligence review*, 1992.

L. Kotthoff, C. Thornton, H. H. Hoos, F. Hutter, and K. Leyton-Brown. Auto-weka 2.0: Automatic model selection and hyperparameter optimization in weka. *Journal of Machine Learning Research*, 2017.

E. LeDell and S. Poirier. H2O AutoML: Scalable automatic machine learning. *ICML Workshop on AutoML*, 2020.

L. Li, H. Wang, L. Zha, Q. Huang, S. Wu, G. Chen, and J. Zhao. Learning a data-driven policy network for pre-training automated feature engineering. *International Conference on Learning Representations*, 2023.

Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 2022.

Z. Li, Q. Zang, D. Ma, J. Guo, T. Zheng, M. Liu, X. Niu, Y. Wang, J. Yang, J. Liu, et al. Autokaggle: A multi-agent framework for autonomous data science competitions. *arXiv preprint arXiv:2410.20424*, 2024.

J. Nam, K. Kim, S. Oh, J. Tack, J. Kim, and J. Shin. Optimized feature generation for tabular data via llms with decision tree reasoning. *Advances in Neural Information Processing Systems*, 2024.

R. S. Olson and J. H. Moore. Tpot: A tree-based pipeline optimization tool for automating machine learning. *ICML Workshop on AutoML*, 2016.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 2011.

H. Pham, M. Guan, B. Zoph, Q. Le, and J. Dean. Efficient neural architecture search via parameters sharing. *International Conference on Machine Learning*, 2018.

L. Prokhorenkova, G. Gusev, A. Vorobev, A. V. Dorogush, and A. Gulin. Catboost: unbiased boosting with categorical features. *Advances in Neural Information Processing Systems*, 2018.

E. Real, A. Aggarwal, Y. Huang, and Q. V. Le. Regularized evolution for image classifier architecture search. *AAAI Conference on Artificial Intelligence*, 2019.

S. Schmidgall, Y. Su, Z. Wang, X. Sun, J. Wu, X. Yu, J. Liu, Z. Liu, and E. Barsoum. Agent laboratory: Using llm agents as research assistants. *arXiv preprint arXiv:2501.04227*, 2025.

Y. Shen, K. Song, X. Tan, D. Li, W. Lu, and Y. Zhuang. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. *Advances in Neural Information Processing Systems*, 2023.

M. Tan and Q. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. *International Conference on Machine Learning*, 2019.

G. Team, P. Georgiev, V. I. Lei, R. Burnell, L. Bai, A. Gulati, G. Tanzer, D. Vincent, Z. Pan, S. Wang, et al. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*, 2024.

H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

G. Wang, Y. Xie, Y. Jiang, A. Mandlekar, C. Xiao, Y. Zhu, L. Fan, and A. Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv: Arxiv-2305.16291*, 2023.

X. Wang, B. Li, Y. Song, F. F. Xu, X. Tang, M. Zhuge, J. Pan, Y. Song, B. Li, J. Singh, et al. Openhands: An open platform for ai software developers as generalist agents. *International Conference on Learning Representations*, 2024.

I. Watson and F. Marir. Case-based reasoning: A review. *The knowledge engineering review*, 1994.

S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao. React: Synergizing reasoning and acting in language models. *International Conference on Learning Representations*, 2023.

Z. You, Y. Zhang, D. Xu, Y. Lou, Y. Yan, W. Wang, H. Zhang, and Y. Huang. Datawiseagent: A notebook-centric llm agent framework for automated data science. *arXiv preprint arXiv:2503.07044*, 2025.

T. Zhang, Z. A. Zhang, Z. Fan, H. Luo, F. Liu, Q. Liu, W. Cao, and L. Jian. Openfe: Automated feature generation with expert-level performance. *International Conference on Machine Learning*, 2023.

B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. *International Conference on Learning Representations*, 2017.

Appendix

A. Prompts for MLE-STAR

A.1. Retriever agent

```
# Competition
{task description}

# Your task
- List {M} recent effective models and their example codes to win the above competition.

# Requirement
- The example code should be concise and simple.
- You must provide an example code, i.e., do not just mention GitHubs or papers.

Use this JSON schema:
Model = {'model_name': str, 'example_code': str}
Return: list[Model]
```

Figure 9 | Prompt used for retrieving task-specific models using web search.

MLE-STAR starts by generating an initial solution. Here, we propose using web search as a tool for MLE-STAR first to retrieve M state-of-the-art models for the given task. Specifically, MLE-STAR leverages a retriever agent $\mathcal{A}_{\texttt{retriever}}$ with the above prompt (Figure 9). $\mathcal{A}_{\texttt{retriever}}$ takes task description $\mathcal{T}_{\texttt{task}}$ as input and retrieves M pairs of $\{\mathcal{T}_{\texttt{model}}, \mathcal{T}_{\texttt{code}}\}$. Here, we guide MLE-STAR to generate the retrieved result as structured output (*i.e.*, JSON). After we obtain JSON file, we parse them into separate model cards.[2]

[2] See `example_intermediate_outputs/retriever_output.txt` in https://github.com/jaehyun513/MLE-STAR.

A.2. Candidate evaluation agent

```
# Introduction
- You are a Kaggle grandmaster attending a competition.
- We will now provide a task description and a model description.
- You need to implement your Python solution using the provided model.

# Task description
{task description}

# Model description
## Model name
{model description}

## Example Python code
{example code}

# Your task
- Implement the solution in Python.
- You must use the model as described in the model description.
- This first solution design should be relatively simple, without ensembling or
hyper-parameter optimization.
- Propose an evaluation metric that is reasonable for this task.
- All the provided data is already prepared and available in the `./input` directory. There
is no need to unzip any files.
- Do not include other models that are not directly related to the model described.
- Use PyTorch rather than TensorFlow. Use CUDA if you need. All the necessary libraries are
installed.
- The code should implement the proposed solution and print the value of the evaluation
metric computed on a hold-out validation set.
- Only use the provided train data in the `./input` directory. Do not load test data.
- If there are more than 30,000 training samples, you must subsample to 30,000 for a faster
run.

# Required
- There should be no additional headings or text in your response.
- Print out or return a final performance metric in your answer in a clear format with the
exact words: 'Final Validation Performance: {final_validation_score}'.
- The code should be a single-file Python program that is self-contained and can be
executed as-is.
- Your response should only contain a single code block.
- Do not use exit() function in the Python code.
- Do not use try: and except: or if else to ignore unintended behavior.
```

Figure 10 | Prompt used for evaluating retrieved models.

MLE-STAR uses candidate evaluation agent $\mathcal{A}_{\texttt{init}}$ to evaluate the performance of the retrieved model. As shown in Figure 10, by taking task description ($\mathcal{T}_{\texttt{task}}$), model description ($\mathcal{T}_{\texttt{model}}$), and corresponding code example ($\mathcal{T}_{\texttt{code}}$), $\mathcal{A}_{\texttt{init}}$ generates a Python script.[3] The Python script for the retrieved model evaluation is guided to be relatively simple, and to contain the evaluation result computed on a hold-out validation set. In addition, if there are too many training samples, $\mathcal{A}_{\texttt{init}}$ uses the subset of training sample for faster execution.

[3] See `example_intermediate_outputs/candidate_evaluation.py` in https://github.com/jaehyun513/MLE-STAR for an example.

A.3. Merging agent

```
# Introduction
- You are a Kaggle grandmaster attending a competition.
- We will now provide a base solution and an additional reference solution.
- You need to implement your Python solution by integrating reference solution to the base
solution.

# Base solution
{base code}

# Reference solution
{reference code}

# Your task
- Implement the solution in Python.
- You have to integrate the reference solution to the base solution.
- Your code base should be the base solution.
- Try to train additional model of the reference solution.
- When integrating, try to keep code with similar functionality in the same place (e.g.,
all preprocessing should be done and then all training).
- When integrating, ensemble the models.
- The solution design should be relatively simple.
- The code should implement the proposed solution and print the value of the evaluation
metric computed on a hold-out validation set.
- Only use the provided train data in the `./input` directory.
- If there are more than 30,000 training samples, you must subsample to 30,000 for a faster
run.

# Required
- There should be no additional headings or text in your response.
- Print out or return a final performance metric in your answer in a clear format with the
exact words: 'Final Validation Performance: {final_validation_score}'.
- The code should be a single-file Python program that is self-contained and can be
executed as-is.
- Your response should only contain a single code block.
- Do not use exit() function in the Python code.
- Do not use try: and except: or if else to ignore unintended behavior
```

Figure 11 | Prompt used for merging the candidate models for generating initial solution.

MLE-STAR leverages an agent $\mathcal{A}_{\texttt{merger}}$ to merge the retrieved models into a consolidated initial solution. As shown in Figure 11, this process is done sequentially, where the prompt guides the agent to integrate the reference code (*i.e.*, the best candidate model code among the models that are not merged yet) into the base code (*i.e.*, the current candidate merged script). The output of $\mathcal{A}_{\texttt{merger}}$ is a Python script[4], which will be the next candidate merged script. See Appendix B for the sequential procedure of MLE-STAR when generating the initial solution.

[4]See `example_intermediate_outputs/merged_candidate.py` in https://github.com/jaehyun513/MLE-STAR for an example.

A.4. Ablation study agent

```
# Introduction
- You are a Kaggle grandmaster attending a competition.
- In order to win this competition, you need to perform an ablation study on the current
Python solution to know which parts of the code contribute the most to the overall
performance.
- We will now provide a current Python solution.
- We will also provide the summaries of previous ablation studies.

# Python solution
{solution script}

## Previous ablation study result {0}
{previous_ablations[0]}

## Previous ablation study result {1}
{previous_ablations[1]}

...

## Previous ablation study result {t-1}
{previous_ablations[t-1]}

# Instructions
- You need you to generate a simple Python code that performs an ablation study on the
train.py script.
- The generated code should create variations by modifying or disabling parts (2-3 parts)
of the training process.
- Your ablation study should concentrate on the other parts that have not been previously
considered.
- For each ablation, print out how the modification affects the model's performance.

# Response format
- There should be no additional headings or text in your response.
- The Python code for the ablation study should not load test data. It should only focus on
training and evaluating the model on the validation set.
- The code should include a printing statement that shows the performance of each ablation.
- The code should consequently print out what part of the code contributes the most to the
overall performance.
```

Figure 12 | Prompt used for generating a Python script for ablation studies.

To effectively explore specialized improvement strategies, MLE-STAR identifies and targets specific code blocks. This code block selection is guided by an ablation study performed by an agent $\mathcal{A}_{\texttt{abl}}$. As shown in Figure 12, $\mathcal{A}_{\texttt{abl}}$ generates a Python code designed to perform an ablation study on current solution. The prompt guides the agent to modify or disable specific component.[5] Moreover, to encourage exploration of different pipeline parts, the summaries of previous ablation studies are also used as valuable feedback. See Appendix C for the example output of the agent $\mathcal{A}_{\texttt{abl}}$.

[5]We provide an example generated code for ablation study (which is generated by $\mathcal{A}_{\texttt{abl}}$) in https://github.com/jaehyun513/MLE-STAR (see `example_intermediate_outputs/ablation.py`).

A.5. Ablation study summarization agent

```
# Your code for ablation study was:
{code for ablation study}

# Ablation study results after running the above code:
{raw result}

# Your task
- Summarize the result of ablation study based on the code and printed output.
```

Figure 13 | Prompt used for summarizing the result of the ablation study.

After executing the code for an ablation study, denoted as a_t, the output result r_t is produced. Since r_t often contains content unrelated to the ablation (for example, printing the loss value across training epochs), a summarization module $\mathcal{A}_{\texttt{summarize}}$ is utilized with the prompt mentioned above (Figure 13). This module takes a_t and r_t as input to summarize and parse the ablation study results. Here, a_t is also used because it provides information about the modification. See Appendix C for the examples of r_t and the summarization result.

A.6. Extractor

```
# Introduction
- You are a Kaggle grandmaster attending a competition.
- In order to win this competition, you need to extract a code block from the current
Python solution and improve the extracted block for better performance.
- Your suggestion should be based on the ablation study results of the current Python
solution.
- We will now provide the current Python solution and the ablation study results.
- We also provide code blocks which you have tried to improve previously.

# Python solution
{solution script}

# Ablation study results
{summary of ablation study}

## Code block {0}
{prev_code_blocks[0]}
## Code block {1}
{prev_code_blocks[1]}
...
## Code block {t-1}
{prev_code_blocks[t-1]}

# Your task
- Given the ablation study results, suggest an effective next plan to improve the above
Python script.
- The plan should be a brief outline/sketch of your proposed solution in natural language
(3-5 sentences).
- Please avoid plan which can make the solution's running time too long (e.g., searching
hyperparameters in a very large search space).
- Try to improve the other part which was not considered before.
- Also extract the code block from the above Python script that need to be improved
according to the proposed plan. You should try to extract the code block which was not
improved before.

# Response format
- Your response should be a brief outline/sketch of your proposed solution in natural
language (3-5 sentences) and a single markdown code block which is the code block that need
to be improved.
- The code block can be long but should be exactly extracted from the Python script
provided above.

Use this JSON schema:

Refine_Plan = {'code_block': str, 'plan': str}
Return: list[Refine_Plan]
```

Figure 14 | Prompt used for extracting the code block that has the most significant impact.

MLE-STAR uses an extractor module $\mathcal{A}_{\texttt{extractor}}$ to analyze the $\mathcal{T}_{\texttt{abl}}$ and then identify the code block c_t. As shown in Figure 14, $\mathcal{A}_{\texttt{extractor}}$ takes the summary of the ablation study, current solution code, and the previously refined code blocks as input, and is guided to output the code block which has the most significant impact on performance. Here, the initial plan for refining the extracted code block is also generated.[6]

[6]See `example_intermediate_outputs/code_block.txt` in https://github.com/jaehyun513/MLE-STAR for an example of extracted code block.

A.7. Coder

```
# Introduction
- You are a Kaggle grandmaster attending a competition.
- In order to win this competition, you need refine the code block for better performance
based on the improvement plan.
- We will now provide the code block and the improvement plan.

# Code block
{code_block}

# Improvement plan
{plan}

# Your task
- Implement the improvement plan on the above code block. But do not remove subsampling if
exists.
- The code block should be improved according to the proposed plan.
- Note that all the variable including actual data is defined earlier (since you are just
seeing a code block), therefore do not introduce dummy variables.

# Response format
- Your response should be a single markdown code block (wrapped in ```) which is the
improved code block.
- There should be no additional headings or text in your response.
```

Figure 15 | Prompt used for implementing refinement plan on the extracted code block.

The implementation of code block refinement is done by $\mathcal{A}_{\texttt{coder}}$, which takes the extracted code block and the refinement plan as input, and outputs the refined code block.[7]

[7]We provide an example of the target code block, proposed plan, and the output of refined code block by $\mathcal{A}_{\texttt{coder}}$ in https://github.com/jaehyun513/MLE-STAR (see `example_intermeidate_outputs/coder_outputs/` directory).

A.8. Planner

```
# Introduction
- You are a Kaggle grandmaster attending a competition.
- In order to win this competition, you have to improve the code block for better
performance.
- We will provide the code block you are improving and the improvement plans you have
tried.

# Code block
{code block}

# Improvement plans you have tried

## Plan: {plans[0]}
## Score: {scores[0]}

## Plan: {plans[1]}
## Score: {scores[1]}

...

## Plan: {plans[k-1]}
## Score: {scores[k-1]}

# Your task
- Suggest a better plan to improve the above code block.
- The suggested plan must be novel and effective.
- Please avoid plans which can make the solution's running time too long (e.g., searching
hyperparameters in a very large search space).
- The suggested plan should be differ from the previous plans you have tried and should
receive a higher score.

# Response format
- Your response should be a brief outline/sketch of your proposed solution in natural
language (3-5 sentences).
- There should be no additional headings or text in your response.
```

Figure 16 | Prompt used for generating the next refinement plan which targets the extracted code block.

To discover potentially more effective or novel refinement strategies (targeting the extracted code block), MLE-STAR iteratively generates further plans through a planning agent $\mathcal{A}_{\texttt{planner}}$. As shown in Figure 16, $\mathcal{A}_{\texttt{planner}}$ takes the extracted code block and the previous attempts as input and proposes the next plan. These are examples of proposed plans:

```
f'''Since feature engineering had the biggest impact, I will focus on improving the cabin
↪  feature extraction. Instead of simply splitting the Cabin string, I will create dummy
↪  variables for each unique Deck and Side. Also, the Cabin_num will be kept as
↪  numerical, imputing missing values using a median strategy to handle potential
↪  outliers. This approach should provide more granular information to the models.'''
```

```
f'''Instead of one-hot encoding 'Deck' and 'Side' directly, I will explore interaction
↪  features between 'Deck', 'Side', and potentially 'Cabin_num'. Specifically, I'll
↪  create combined features like 'Deck_Side' and 'Deck_Cabin_num' to capture potential
↪  dependencies. Furthermore, I will impute missing 'Cabin_num' values using a more
↪  sophisticated method like k-NN imputation, considering other features like 'Deck',
↪  'Side', and 'RoomService' to improve imputation accuracy. This should capture more
↪  complex relationships within the cabin data and lead to better model performance.'''
```

```
f'''I propose a plan that focuses on a more nuanced approach to 'Cabin_num' and
↪  interaction terms. First, I'll bin 'Cabin_num' into ordinal categories (e.g., low,
↪  medium, high) based on quantiles, as the absolute number might not be as important as
↪  its relative position. Then, I'll create interaction features between the binned
↪  'Cabin_num', 'Deck', and 'Side' using one-hot encoding. This will allow the model to
↪  learn specific combinations of cabin location and number range that might be
↪  predictive. Finally, I will use a simple imputer for the missing values in
↪  'Cabin_num' before binning.'''
```

A.9. Ensemble strategy planner

```
# Introduction
- You are a Kaggle grandmaster attending a competition.
- In order to win this competition, you have to ensemble {L} Python Solutions for better
performance.
- We will provide the Python Solutions and the ensemble plans you have tried.

# 1st Python Solution
{solution1}
# 2nd Python Solution
{solution2}
...
# {L}th Python Solution
{solutionL}

# Ensemble plans you have tried

## Plan: {plans[0]}
## Score: {scores[0]}
## Plan: {plans[1]}
## Score: {scores[1]}
...
## Plan: {plans[r-1]}
## Score: {scores[r-1]}

# Your task
- Suggest a better plan to ensemble the {L} solutions. You should concentrate how to merge,
not the other parts like hyperparameters.
- The suggested plan must be easy to implement, novel, and effective.
- The suggested plan should be differ from the previous plans you have tried and should
receive a higher (or lower) score.

# Response format
- Your response should be an outline/sketch of your proposed solution in natural language.
- There should be no additional headings or text in your response.
- Plan should not modify the original solutions too much since execution error can occur.
```

Figure 17 | Prompt used for generating the next ensemble plan.

As shown in Figure 17, similar to $\mathcal{A}_{\texttt{planner}}$, $\mathcal{A}_{\texttt{ens\_planner}}$ proposes an effective ensemble plan based on the history of previously attempted ensemble plans and their resulting performance as feedback. These are examples of attempted ensemble plans.

```
f'''Averaging the predicted probabilities from both models is a straightforward and
↪  effective ensembling technique. First, modify the AutoGluon solution to output
↪  probabilities instead of hard predictions using `predictor.predict_proba(test_data)`.
↪  Then, obtain the predicted probabilities from the LightGBM model using
↪  `lgbm_classifier.predict_proba(X_test_processed)`. Average these probabilities for
↪  each class. Finally, generate the final predictions by thresholding the averaged
↪  probability of the 'Transported' class at 0.5. Create the submission file based on
↪  these averaged and thresholded predictions.'''
```

```
f'''Here's an ensembling plan leveraging stacking with a simple meta-learner:

1.  **Generate Predictions:** Use both the AutoGluon model and the LightGBM model to
↪  generate predictions on the original training data (train.csv).  This is crucial for
↪  training the meta-learner.  For AutoGluon, use `predictor.predict_proba(train_data)`
↪  and extract the probabilities for the 'Transported' class.  For LightGBM, preprocess
↪  the training data using the same pipeline as the test data and get probabilities with
↪  `lgbm_classifier.predict_proba(X_processed)` and again, extract the probabilities for
↪  the 'Transported' class.

2.  **Create Meta-Features:** Combine the predicted probabilities from AutoGluon and
↪  LightGBM for the training data into a new dataframe. This dataframe will have two
↪  columns: 'AutoGluon_Prob' and 'LGBM_Prob', and the 'Transported' column from the
↪  original training data as the target variable for the meta-learner.

3.  **Train Meta-Learner:** Use a simple model like Logistic Regression as the
↪  meta-learner. Train this Logistic Regression model using the meta-features
↪  (AutoGluon_Prob, LGBM_Prob) to predict the 'Transported' column.  This step aims to
↪  learn how to best combine the predictions of the base models.

4.  **Generate Test Predictions:** Get the predicted probabilities from AutoGluon and
↪  LightGBM on the test set, as in the averaging approach.

5.  **Create Meta-Features for Test Data:** Create a dataframe for the test data, with
↪  the same structure as the training meta-features (AutoGluon_Prob, LGBM_Prob) from the
↪  test set.

6.  **Meta-Learner Prediction:** Use the trained Logistic Regression model to predict the
↪  final 'Transported' probabilities on the test meta-features.

7.  **Threshold and Submit:** Threshold the predicted probabilities from the meta-learner
↪  at 0.5 to get the final predictions (True/False) and create the submission file.'''
```

```
f'''Here's an ensembling plan that focuses on weighted averaging with optimized weights
↪  determined by a simple grid search on a validation set:

1.  **Validation Split:** Split the original training data into two parts: a training set
↪  (e.g., 80% of the data) and a validation set (e.g., 20% of the data).  Crucially,
↪  perform the preprocessing steps (OneHotEncoding, Scaling, etc.) separately on the
↪  training and validation sets to avoid data leakage.

2.  **Generate Validation Predictions:** Use both the AutoGluon model and the LightGBM
↪  model to generate predictions on the validation set. For AutoGluon, obtain
↪  probabilities using `predictor.predict_proba(validation_data)`.  For LightGBM,
↪  preprocess the validation data using the same pipeline trained on the training split
↪  and get probabilities using `lgbm_classifier.predict_proba(X_validation_processed)`.

3.  **Grid Search for Optimal Weights:** Define a grid of weights for AutoGluon and
↪  LightGBM. For instance, iterate through weights from 0.0 to 1.0 in increments of 0.1
↪  for AutoGluon, with the LightGBM weight being (1 - AutoGluon weight). For each weight
↪  combination:
    *   Calculate the weighted average of the predicted probabilities from AutoGluon and
    ↪  LightGBM on the validation set.
    *   Threshold the averaged probabilities at 0.5 to obtain binary predictions.
    *   Calculate the accuracy of these predictions against the true labels in the
    ↪  validation set.

4.  **Select Best Weights:** Choose the weight combination that yields the highest
↪  accuracy on the validation set.

5.  **Generate Test Predictions:** Obtain the predicted probabilities from AutoGluon and
↪  LightGBM on the test set, as before.

6.  **Weighted Averaging on Test Set:** Use the optimal weights determined in step 4 to
↪  calculate the weighted average of the predicted probabilities from AutoGluon and
↪  LightGBM on the test set.

7.  **Threshold and Submit:** Threshold the weighted average probabilities at 0.5 to
↪  obtain the final predictions and create the submission file.

This plan is easy to implement, avoids complex meta-learners that can overfit, and
↪  focuses on finding the best combination of the two models based on a validation set.
↪  It adapts to the strengths of each model by giving them different weights.'''
```

A.10. Ensembler

```
# Introduction
- You are a Kaggle grandmaster attending a competition.
- In order to win this competition, you need to ensemble {L} Python Solutions for better
performance based on the ensemble plan.
- We will now provide the Python Solutions and the ensemble plan.

# 1st Python Solution
{solution1}
# 2nd Python Solution
{solution2}
...
# {L}th Python Solution
{solutionL}

# Ensemble Plan
{plan}

# Your task
- Implement the ensemble plan with the provided solutions.
- Unless mentioned in the ensemble plan, do not modify the original Python Solutions too
much."
- All the provided data (except previous submissions; do not load submissions) is already
prepared and available in the `.\input` directory. There is no need to unzip any files.
- The code should implement the proposed solution and print the value of the evaluation
metric computed on a hold-out validation set.

# Response format required
- Your response should be a single markdown code block (wrapped in ```) which is the
ensemble of {L} Python Solutions.
- There should be no additional headings or text in your response.
- Do not subsample or introduce dummy variables. You have to provide full new Python
Solution using the {L} provided solutions.
- Do not forget the `./final/submission.csv` file.
- Print out or return a final performance metric in your answer in a clear format with the
exact words: 'Final Validation Performance: {final_validation_score}'.
- The code should be a single-file Python program that is self-contained and can be
executed as-is.
```

Figure 18 | Prompt used for implementing ensemble plan on the solutions generated by MLE-STAR in parallel.

The proposed ensemble plan is implemented by $\mathcal{A}_{\texttt{ensembler}}$. This agent takes the two final solutions which is generated in parallel by MLE-STAR, and the ensemble plan as input, and outputs the Python script, *i.e.*, the merged code solution (see Appendix C for examples since the final solution is selected among the merged code solution).

A.11. Debugging agent

```
# Code with an error:
{code}

# Error:
{bug}

# Your task
- Please revise the code to fix the error.
- Do not remove subsampling if exists.
- Provide the improved, self-contained Python script again.
- There should be no additional headings or text in your response.
- All the provided input data is stored in "./input" directory.
- Remember to print a line in the code with 'Final Validation Performance:
{final_validation_score}' so we can parse performance.
- The code should be a single-file python program that is self-contained and can be
executed as-is.
- Your response should only contain a single code block.
- Do not use exit() function in the refined Python code.
```

Figure 19 | Prompt used for debugging.

If the execution of a Python script triggers an error, MLE-STAR employs a debugging module $\mathcal{A}_{\texttt{debugger}}$ to attempt correction using the above prompt (Figure 19).

A.12. Data leakage checker

```
# Python code
{code}

# Your task
- Extract the code block where the validation and test samples are preprocessed using
training samples.
- Check that the model is trained with only training samples.
- Check that before printing the final validation score, the model is not trained the
validation samples.
- Also check whether the validation and test samples are preprocessed correctly, preventing
information from the validation or test samples from influencing the training process
(i.e., preventing data leakage).


# Requirement
- Extract a code block and also check the data leakage.
- The code block should be an exact subset of the above Python code.
- Your response for a code block should be a single markdown code block.
- If data leakage is present on validation and test samples, answer 'Yes Data Leakage'.
- If data leakage is not present on validation and test samples, answer 'No Data Leakage'.

Use this JSON schema:
Answer = {'leakage_status': str, 'code_block': str}
Return: list[Answer]
```

Figure 20 | Prompt used for extract the code block whether data preprocessing is done.

```
# Python code
{code}

# Your task
- In the above Python code, the validation and test samples are influencing the training
process, i.e., not correctly preprocessed.
- Ensure that the model is trained with only training samples.
- Ensure that before printing the final validation score, the model is not trained on the
validation samples.
- Refine the code to prevent such data leakage problem.

# Requirement
- Your response should be a single markdown code block.
- Note that all the variables are defined earlier. Just modify it with the above code.
```

Figure 21 | Prompt used for correcting the code block with a risk of data leakage.

To mitigate the risk of introducing data leakage, MLE-STAR first extract the code block where preprocessing is done. This is achieved by using the above prompt in Figure 20, which takes the current solution script as input, and then generates (1) the code block and (2) whether the extracted code block has a risk of data leakage. If leakage is detected, the code block is corrected with the prompt in Figure 21, and MLE-STAR replaces the original code block to the corrected version.

A.13. Data usage checker

```
I have provided Python code for a machine learning task (attached below):

# Solution Code
{initial solution}

Does above solution code uses all the information provided for training? Here is task
description and some guide to handle:

# Task description
{task description}

# Your task
- If the above solution code does not use the information provided, try to incorporate all.
Do not bypass using try-except.
- DO NOT USE TRY and EXCEPT; just occur error so we can debug it!
- See the task description carefully, to know how to extract unused information
effectively.
- When improving the solution code by incorporating unused information, DO NOT FORGET to
print out 'Final Validation Performance: {final_validation_score}' as in original solution
code.

# Response format:
Option 1: If the code did not use all the provided information, your response should be a
single markdown code block (wrapped in ```) which is the improved code block. There should
be no additional headings or text in your response
Option 2: If the code used all the provided information, simply state that "All the
provided information is used.
```

Figure 22 | Prompt used for data usage checker.

To ensure the utilization of all relevant provided data, MLE-STAR utilizes a data usage checker agent $\mathcal{A}_{\texttt{data}}$. This agent checks the initial solution with the task description, and revise the initial script using the prompt in Figure 22.

B. Algorithms

B.1. Algorithm for generating an initial solution

Algorithm 1 Generating an initial solution

Input: task description $\mathcal{T}_{\texttt{task}}$, datasets $\mathcal{D}$, score function h, number of retrieved models M,
$\{\mathcal{T}^i_{\texttt{model}}, \mathcal{T}^i_{\texttt{code}}\}_{i=1}^{M} = \mathcal{A}_{\texttt{retriever}}(\mathcal{T}_{\texttt{task}})$
for $i = 1$ to M **do**
    $s^i_{\texttt{init}} = \mathcal{A}_{\texttt{init}}(\mathcal{T}_{\texttt{task}}, \mathcal{T}^i_{\texttt{model}}, \mathcal{T}^i_{\texttt{code}})$
    Evaluate $h(s^i_{\texttt{init}})$ using $\mathcal{D}$
end for
$s_0 \leftarrow s^{\pi(1)}_{\texttt{init}}$
$h_{\texttt{best}} \leftarrow h(s_0)$
for $i = 2$ to M **do**
    $s_{\texttt{candidate}} \leftarrow \mathcal{A}_{\texttt{merger}}(s_0, s^{\pi(i)}_{\texttt{init}})$
    Evaluate $h(s_{\texttt{candidate}})$ using $\mathcal{D}$
    if $h(s_{\texttt{candidate}}) \geq h_{\texttt{best}}$ **then**
        $s_0 \leftarrow s_{\texttt{candidate}}$
        $h_{\texttt{best}} \leftarrow h(s_0)$
    else
        break
    end if
end for
Output: initial solution s_0

B.2. Algorithm for refining a code block for solution improvement

Algorithm 2 Refining solution

Input: initial solution s_0, outer loop steps T, inner loop steps K
$s_{\texttt{final}} \leftarrow s_0$
$h_{\texttt{best}} \leftarrow h(s_0)$
$\mathcal{T}_{\texttt{abl}}, C = \{\}, \{\}$
for $t = 0$ to $T - 1$ **do**
  $a_t = \mathcal{A}_{\texttt{abl}}(s_t, \mathcal{T}_{\texttt{abl}})$
  $r_t = \texttt{exec}(a_t)$
  $\mathcal{T}^t_{\texttt{abl}} = \mathcal{A}_{\texttt{summarize}}(a_t, r_t)$
  $c_t, p_0 = \mathcal{A}_{\texttt{extractor}}(\mathcal{T}^t_{\texttt{abl}}, s_t, C)$
  $c^0_t = \mathcal{A}_{\texttt{coder}}(c_t, p_0)$
  $s^0_t = s_t.\texttt{replace}(c_t, c^0_t)$
  Evaluate $h(s^0_t)$ using $\mathcal{D}$
  if $h(s^0_t) \geq h_{\texttt{best}}$ **then**
    $s_{\texttt{final}} \leftarrow s^0_t$
    $h_{\texttt{best}} \leftarrow h(s^0_t)$
  end if
  for $k = 1$ to $K - 1$ **do**
    $p_k = \mathcal{A}_{\texttt{planner}}(c_t, \{p_j, h(s^j_t)\}^{k-1}_{j=0})$
    $c^k_t = \mathcal{A}_{\texttt{coder}}(c_t, p_k)$
    $s^k_t = s_t.\texttt{replace}(c_t, c^k_t)$
    Evaluate $h(s^k_t)$ using $\mathcal{D}$
    if $h(s^k_t) \geq h_{\texttt{best}}$ **then**
      $s_{\texttt{final}} \leftarrow s^k_t$
      $h_{\texttt{best}} \leftarrow h(s^k_t)$
    end if
  end for
  $\mathcal{T}_{\texttt{abl}} \leftarrow \mathcal{T}_{\texttt{abl}} + \mathcal{T}^t_{\texttt{abl}}$
  $C \leftarrow C + c_t$
end for
Output: final solution $s_{\texttt{final}}$

B.3. Algorithm for further improvement by exploring ensemble strategies

Algorithm 3 Ensembling final solutions

Input: candidate final solutions $s^1_{\texttt{final}}, \cdots, s^L_{\texttt{final}}$, ensemble loop steps R
$e_0 = \mathcal{A}_{\texttt{ens\_planner}}(\{s^l_{\texttt{final}}\}_{l=1}^L)$
$s^0_{\texttt{ens}} = \mathcal{A}_{\texttt{ensembler}}(e_0, \{s^l_{\texttt{final}}\}_{l=1}^L)$
Evaluate $h(s^0_{\texttt{ens}})$ using $\mathcal{D}$
for $r = 1$ to $R - 1$ **do**
    $e_r = \mathcal{A}_{\texttt{ens\_planner}}(\{s^l_{\texttt{final}}\}_{l=1}^L, \{(e_j, h(s^j_{\texttt{ens}}))\}_{j=0}^{r-1})$
    $s^r_{\texttt{ens}} = \mathcal{A}_{\texttt{ensembler}}(e_r, \{s^l_{\texttt{final}}\}_{l=1}^L)$
    Evaluate $h(s^r_{\texttt{ens}})$ using $\mathcal{D}$
end for
$s^*_{\texttt{ens}} = s^{r^*}_{\texttt{ens}}$ where $r^* = \arg\max_{r \in \{0,\ldots,R-1\}} h(s^r_{\texttt{ens}})$
Output: $s^*_{\texttt{ens}}$

C. Qualitative examples

C.1. Generated code for ablation study

We provide an example generated code for ablation study (which is generated by $\mathcal{A}_{\texttt{abl}}$) in the supplementary material (see `example_outputs/ablation.py`).

C.2. Raw output of ablation study after execution

```
[LightGBM] [Info] Number of positive: 2854, number of negative: 2709
[LightGBM] [Info] Auto-choosing col-wise multi-threading, the overhead of testing was 0.001167 seconds.
You can set `force_col_wise=true` to remove the overhead.
[LightGBM] [Info] Total Bins 1647
[LightGBM] [Info] Number of data points in the train set: 5563, number of used features: 26
[LightGBM] [Info] [binary:BoostFromScore]: pavg=0.513033 -> initscore=0.052142
[LightGBM] [Info] Start training from score 0.052142
Baseline Validation Performance: 0.8195542774982028
[LightGBM] [Info] Number of positive: 2854, number of negative: 2709
[LightGBM] [Info] Auto-choosing col-wise multi-threading, the overhead of testing was 0.002991 seconds.
You can set `force_col_wise=true` to remove the overhead.
[LightGBM] [Info] Total Bins 1647
[LightGBM] [Info] Number of data points in the train set: 5563, number of used features: 26
[LightGBM] [Info] [binary:BoostFromScore]: pavg=0.513033 -> initscore=0.052142
[LightGBM] [Info] Start training from score 0.052142
Ablation 1 (No StandardScaler) Validation Performance: 0.8102084831056794
[LightGBM] [Info] Number of positive: 2854, number of negative: 2709
[LightGBM] [Info] Auto-choosing col-wise multi-threading, the overhead of testing was 0.000367 seconds.
You can set `force_col_wise=true` to remove the overhead.
[LightGBM] [Info] Total Bins 1609
[LightGBM] [Info] Number of data points in the train set: 5563, number of used features: 7
[LightGBM] [Info] [binary:BoostFromScore]: pavg=0.513033 -> initscore=0.052142
[LightGBM] [Info] Start training from score 0.052142
Ablation 2 (No OneHotEncoder) Validation Performance: 0.7886412652767792
[LightGBM] [Info] Number of positive: 2854, number of negative: 2709
[LightGBM] [Info] Auto-choosing col-wise multi-threading, the overhead of testing was 0.001942 seconds.
You can set `force_col_wise=true` to remove the overhead.
[LightGBM] [Info] Total Bins 1647
[LightGBM] [Info] Number of data points in the train set: 5563, number of used features: 26
[LightGBM] [Info] [binary:BoostFromScore]: pavg=0.513033 -> initscore=0.052142
[LightGBM] [Info] Start training from score 0.052142
Ablation 3 (No Imputation) Validation Performance: 0.8195542774982028
Final Validation Performance: 0.8195542774982028
```

Figure 23 | Example output after running the code for ablation study.

We provide an example output after running a code for ablation study using spaceship-titanic competition. As shown in Figure 23, the execution result often contains content unrelated to the ablation (*e.g.*, training information of LightGBM). Therefore, MLE-STAR utilizes $\mathcal{A}_{\texttt{summarize}}$ to parse the ablation study results, which will be illustrated in the following Appendix C.3.

C.3. Summary of ablation study

The ablation study investigated the impact of three preprocessing steps on the performance of a LightGBM classifier: StandardScaler, OneHotEncoder, and Imputation. The baseline model, which included all three preprocessing steps, achieved a validation accuracy of 0.8196.

*** **Ablation 1 (No StandardScaler):** Removing the StandardScaler resulted in a slightly lower validation accuracy of 0.8102. This suggests that scaling the numerical features does contribute positively to the model's performance, although the impact is relatively small.**

*** **Ablation 2 (No OneHotEncoder):** Removing the OneHotEncoder led to a more significant drop in validation accuracy to 0.7886. This indicates that encoding the categorical features using OneHotEncoder is important for the model's ability to learn from the data.**

*** **Ablation 3 (No Imputation):** Removing the Imputation did not change the validation accuracy, which remained at 0.8196. This suggests that the imputation strategy used (mean for numerical and mode for categorical) does not hurt the model performance.**

In summary, the OneHotEncoder has the most significant positive impact on the model's performance, followed by the StandardScaler. The Imputation does not seem to have a significant impact on the model's performance.

Figure 24 | Example of summarized result of ablation study.

To parse the information only about the impact of each ML components, MLE-STAR leverages $\mathcal{A}_{\texttt{summarize}}$ to summarize the raw output of ablation study such as Figure 23. As a result, we obtain the well-organized summary of the ablation study as shown in Figure 24. Note that such summarization is used as input of $\mathcal{A}_{\texttt{extractor}}$ to extract the code block which has most significant impact on performance.

D. Qualitative comparison

We provide qualitative comparison results (*i.e.*, the final solution code) in https://github.com/jaehyun513/MLE-STAR (see `example_final_solutions/` directory). Solutions generated by MLE-STAR is denoted as `mle_star.py` and solutions generated by AIDE (Jiang et al., 2025) is denoted as `aide.py` in folder name with competition ID. Note that both agent used Gemini-2.0-Flash as a base LLM.

E. Benchmark

E.1. MLE-bench Lite

Table 7 | Competitions contained in MLE-bench Lite (Chan et al., 2025).

Competition ID	Category	Dataset Size (GB)
aerial-cactus-identification	Image Classification	0.0254
aptos2019-blindness-detection	Image Classification	10.22
denoising-dirty-documents	Image To Image	0.06
detecting-insults-in-social-commentary	Text Classification	0.002
dog-breed-identification	Image Classification	0.75
dogs-vs-cats-redux-kernels-edition	Image Classification	0.85
histopathologic-cancer-detection	Image Regression	7.76
jigsaw-toxic-comment-classification-challenge	Text Classification	0.06
leaf-classification	Image Classification	0.036
mlsp-2013-birds	Audio Classification	0.5851
new-york-city-taxi-fare-prediction	Tabular	5.7
nomad2018-predict-transparent-conductors	Tabular	0.00624
plant-pathology-2020-fgvc7	Image Classification	0.8
random-acts-of-pizza	Text Classification	0.003
ranzcr-clip-catheter-line-classification	Image Classification	13.13
siim-isic-melanoma-classification	Image Classification	116.16
spooky-author-identification	Text Classification	0.0019
tabular-playground-series-dec-2021	Tabular	0.7
tabular-playground-series-may-2022	Tabular	0.57
text-normalization-challenge-english-language	Seq->Seq	0.01
text-normalization-challenge-russian-language	Seq->Seq	0.01
the-icml-2013-whale-challenge-right-whale-redux	Audio Classification	0.29314

In this paper, we utilize MLE-bench (especially Lite version) (Chan et al., 2025) as our main benchmark to verify MLE-STAR's effectiveness compared to the alternatives. In a nutshell, MLE-bench consists of 75 offline Kaggle competitions. Each competition has an associated description, dataset, and grading code. Additionally, MLE-bench consists of various problem types, such as tabular prediction, text classification, image classification, etc. However, since utilizing full 75 competitions is expensive, we use the Lite version, which is the low complexity split of MLE-bench (*i.e.*, MLE-bench Lite). MLE-bench Lite consists of 22 competitions, and the description of competitions is provided in Table 7.

E.2. Tabular tasks from DS-Agent

Table 8 | Tabular competitions used in DS-Agent (Guo et al., 2024)

Competition ID	Category	Evaluation Metrics
media-campaign-cost	Tabular Regression	RMLSE
wild-blueberry-yield	Tabular Regression	MAE
spaceship-titanic	Tabular Classification	Accuracy
enzyme-substrate	Tabular Classification	AUROC

We also provide the descriptions of tabular competitions used in DS-Agent's development phase (Guo et al., 2024) in Table 8.

E.3. Generating submission file

```
# Introduction
- You are a Kaggle grandmaster attending a competition.
- In order to win this competition, you need to come up with an excellent solution in
Python.
- We will now provide a task description and a Python solution.
- What you have to do on the solution is just loading test samples and create a submission
file.

# Task description
{task description}

# Python solution
{final solution}

# Your task
- Load the test samples and create a submission file.
- All the provided data is already prepared and available in the `./input` directory. There
is no need to unzip any files.
- Test data is available in the `./input` directory.
- Save the test predictions in a `submission.csv` file. Put the `submission.csv` into
`./final` directory.
- You should not drop any test samples. Predict the target value for all test samples.
- This is a very easy task because the only thing to do is to load test samples and then
replace the validation samples with the test samples. Then you can even use the full
training set!

# Required
- Do not modify the given Python solution code too much. Try to integrate test submission
with minimal changes.
- There should be no additional headings or text in your response.
- The code should be a single-file Python program that is self-contained and can be
executed as-is.
- Your response should only contain a single code block.
- Do not forget the ./final/submission.csv file.
- Do not use exit() function in the Python code.
- Do not use try: and except: or if else to ignore unintended behavior.
```

Figure 25 | Prompt used for incorporating loading test sample and generating a submission file.

In order to evaluate on MLE-bench Lite, one should create a submission file about prediction results on test samples with required format. To achieve this, MLE-STAR uses an agent $\mathcal{A}_{\texttt{test}}$, which takes the task description and the final solution as input, and outputs the code that incorporates loading test sample and creating a submission file. This is done by using a prompt in Figure 25.

```
# Introduction
- From the give Python solution, you need to extract a code block where subsampling of
training samples is used. We will now provide the current Python solution."

# Current Python solution
{final solution}

# Your task
- Extract a code block where subsampling of training samples is used.

# Response format
- Your response should be a single markdown code block (wrapped in ```) which is the code
block.
- The code block should be exactly extracted from the Python script provided above.
```

Figure 26 | Prompt used for extracting the code block which performs subsampling.

```
# Introduction
- From the give Python code block, remove the subsampling and make it to use full training
samples. We will now provide the current Python code block.

# Current Python code block
{code block with subsampling}

# Your task
- Remove the subsampling and make it to use full training samples.
- Note that all the variable including actual data is defined earlier (since you are just
seeing a code block), therefore do not introduce dummy variables.

# Response format
- Your response should be a single markdown code block (wrapped in ```) which is the code
block.
```

Figure 27 | Prompt used for guiding MLE-STAR to utilizie full training samples.

Removing subsampling. As shown in Figure 10, MLE-STAR uses the subset of training sample for faster refinement (since evaluating the solution candidate can take a lot of time). However, in order to get a better performance, when generating a submission file MLE-STAR removes such subsampling code. Specifically, this is done by first extracting the code block which performs subsampling (using prompt in Figure 26), and then modify the extracted code block to utilize all the provided samples, using prompt in Figure 27.

F. Experimental setup

We conducted our experiments mainly using 96 vCPUs with 360 GB Memory (Intel(R) Xeon(R) CPU), and 8 NVIDIA V100 GPUs with 16 GB Memory.

Required time to generate a single solution using MLE-STAR. With the configuration of four retrieved models, four inner loops, four outer loops, and five rounds for exploring the ensemble strategy, MLE-STAR requires 14.1 hours to generate a single final solution, on average across 22 tasks and all three random trials (*i.e.*, total 66 experiments). On the other hand, we found that AIDE (Jiang et al., 2025) requires 15.4 hours. This indicates that our method does not require more time to run compare to the best alternative. Note that a maximum time limit of 24 hours was set for both methods, following the MLE-bench's experimental setup.

G. Additional quantitative results

Table 9 | Additional comparisons with AutoGluon and DS-Agent in four tabular tasks.

Model	media-campaign-cost	wild-blueberry-yield	spaceship-titanic	enzyme-substrate
Evaluation Metrics	RMLSE (↓)	MAE (↓)	Accuracy (↑)	AUROC (↑)
AutoGluon (Erickson et al., 2020)	0.2707	305	0.8044	0.8683
DS-Agent (Guo et al., 2024)				
gpt-3.5	**0.2702**	291	/	0.5534
gpt-4	0.2947	267	0.7977	0.8322
gemini-2.0-flash	0.2964	213	0.7982	0.8727
MLE-STAR (Ours)				
gemini-2.0-flash	0.2911	**163**	**0.8091**	**0.9101**

This section provides detailed results for the comparison with DS-Agent (Guo et al., 2024). In particular, we provide additional comparisons with AutoGluon (Erickson et al., 2020) and DS-Agent using other LLMs (*i.e.*, GPT-3.5 and GPT-4). Except for DS-Agent with Gemini-2.0-Flash and MLE-STAR, all experimental results are taken from the original paper (Guo et al., 2024). As shown in Table 9, MLE-STAR consistently outperforms DS-Agent with Gemini-2.0-Flash, while also outperforms AutoGluon with high margin on three tabular tasks.

It is worth to note that AutoGluon is restricted to task types, *i.e.*, specially designed for tabular data. In contrast, MLE-STAR is a general framework for any kinds of tasks, where well-written task description, containing the task information, is the only requirement to work on the given tasks. Therefore, while AutoGluon is not a direct competitor in this regard, MLE-STAR shows improved performance even when compared to AutoGluon.

H. Analysis on data contamination

```
Your task is to check whether the python solution is similar to the reference discussion.
Now we will give you reference discussion and our python solution.

# Reference discussion
{reference discussion}

# Python solution
{final solution}

# Your task
- Check whether the python solution just copy and pastes the reference discussion.
- If it is sufficiently novel and different, please answer 'Novel'.
- Otherwise, if you think it is too similar, please answer 'Same'.
- Your answer should be only one of 'Novel' or 'Same'.
```

Figure 28 | Prompt used for identifying whether the final solution generated by MLE-STAR is novel.

Since Kaggle competitions in MLE-bench are publicly accessible, there is a potential risk that LLMs might have been trained with the relevant discussions about the challenge. For example, if an LLM has memorized a discussion of the best performing solution, one easy way for the MLE agent to follow that discussion during the refinement phase.

However, to alleviate such potential issue, we show that MLE-STAR's solution is sufficiently novel compared to the discussions on Kaggle. Here, we use discussions collected in GibHub repository of MLE-bench (Chan et al., 2025) are collected by the authors of MLE-bench (Chan et al., 2025). To be specific, these discussions are top discussion posts of each competition. As a result, we collected a total of 25 discussions from 7 competitions, resulting in 75 discussion-solution pairs, where solution represents the final solution obtained by MLE-STAR. Using LLM as a judge with the prompt in Figure 28, we found that all the final solutions generated by MLE-STAR with Gemini-2.0-Flash were judged to be sufficiently novel compared to the top discussions. Note that we use Gemini-2.5-Pro to judge the novelty of MLE-STAR's solutions.

I. Broader impacts

By automating complex ML tasks, MLE-STAR could lower the barrier to entry for individuals and organizations looking to leverage ML, potentially fostering innovation across various sectors. In addition, as state-of-the-art models are updated and improved over time, the performance of solutions generated by MLE-STAR is expected to be automatically boosted. This is because our framework leverages a search engine to retrieve effective models from the web to form its solutions. This inherent adaptability ensures that MLE-STAR continues to provide increasingly better solutions as the field of ML advances.

J. Related works on data science agents

While our work focuses on LLM-based agents tailored for machine learning engineering, other research explores agents for general data science tasks (Hu et al., 2024; Huang et al., 2024b; Jing et al., 2025), including data analysis and visualization. Among these, Data Interpreter (Hong et al., 2024) employs a graph-based approach, dividing tasks into subtasks and refining the task graph based on successful completion. DatawiseAgent (You et al., 2025) proposes a two-stage process: initially generating a tree-structured plan, followed by an exploration of the solution space. Although these methods exhibit generalizability to various data science tasks, including aspects of machine learning engineering, their evaluation prioritizes overall task completion rates rather than performance on specific engineering challenges.