PiML: Automated Machine Learning Workflow Optimization using LLM Agents

Anonymous[1]

[1]Anonymous Institution

Abstract In this paper, we introduce PiML, a novel automated pipeline specifically designed for solving real-world machine learning (ML) tasks such as Kaggle competitions. PiML integrates iterative reasoning, automated code generation, adaptive memory construction, and systematic debugging to tackle complex problems effectively. To rigorously assess our framework, we selected 26 diverse competitions from the MLE-Bench benchmark, ensuring comprehensive representation across various complexity levels, modalities, competition types, and dataset sizes. We quantitatively compared PiML's performance to AIDE—the best-performing existing baseline from MLE-Bench—across multiple evaluation metrics: Valid Submission rate, Submissions Above Median, Average Percentile Rank, and Medal Achievement Rate. Using the "o3-mini" model, PiML surpassed the baseline in submissions above median (34.61% vs 30.77%), medal attainment rate (26.92% vs 23.08%), and average percentile rank (43.75% vs 39.06%). These results highlight PiML's flexibility, robustness, and superior performance on practical and complex ML challenges.

1 Introduction

Designing an end-to-end machine learning (ML) workflow is a complex effort that requires substantial expertise, as manually crafting and optimizing these workflows for specific tasks is both labor-intensive and knowledge-intensive. This challenge has been partially addressed by AutoML (Erickson et al. (2020a), Tang et al. (2024), Shchur et al. (2023)), which automates various stages of the workflow, streamlining processes that would otherwise demand extensive human effort (Feurer et al. (2015)). However, while AutoML has improved efficiency to a degree, it operates within a predefined rule set and often lacks the flexibility necessary to adapt to the domain specific requirements of the problem (Zöller and Huber (2021)).

In contrast, the emergence of Large Language Models (LLMs) has revolutionized problem-solving approaches thanks to their expansive knowledge bases and reasoning capabilities. Techniques like Chain of Thought (CoT)(Wei et al. (2023)), Tree of Thought (ToT)(Yao et al. (2023a)), and ReAct(Yao et al. (2023b)) have demonstrated the potential of LLMs in tackling complex coding tasks, showcasing their ability to facilitate complex reasoning processes. These capabilities can be applied to ML workflows, offering potential solutions to previously challenging downstream tasks. Many works attempted to address some parts of a ML workflow - feature engineering (Hollmann et al. (2023); Jeong et al. (2024); Zhang et al. (2024b); Gong et al. (2024); Li et al. (2025); Malberg et al. (2024)), Model Selection - (Zhang et al. (2023); Shen et al. (2023)); HPO - (Liu et al. (2025); Zhang et al. (2024a)).

We propose a novel multi-agent framework, *PiML: Automated Machine Learning Workflow Optimization using LLM Agents* for exploring the true exploratory nature of ML problem solving via iterative experimentation. Unlike many other similar works, our framework enables step-by-step approach to problem solving.

Submitted to AutoML 2025 © 2025 the authors, released under CC BY 4.0

Our contributions include:

1. **Automated Agent Pipeline**: We introduce PiML, a structured and iterative automated pipeline that systematically leverages an agent's internal reasoning ("Thoughts") and executable code ("Actions") guided by summarized execution feedback ("Observations") to autonomously solve machine learning tasks from Kaggle competitions.
2. **Adaptive Memory Management**: A novel multi-tier memory construction strategy, effectively balancing detailed recent context with summarized historical actions.
3. **Robust Error Handling via Debug Chain**: We present a structured and systematic "Debug Chain" mechanism that iteratively refines erroneous code actions generated by the pipeline, improving error resolution and enabling efficient self-correction without human supervision.
4. **Experimental Validation and Competitiveness**: Empirical evaluation on the diverse MLE-Bench(MLE-Pi) dataset demonstrates the flexibility and effectiveness of PiML. Specifically, our results indicate superior or competitive performance against strong baseline automated frameworks, thereby highlighting PiML's potential to autonomously achieve competitive results across various competition complexities and categories.

2 Methodology

In this section, we describe our automated agent pipeline designed to solve input ML tasks such as Kaggle competitions.

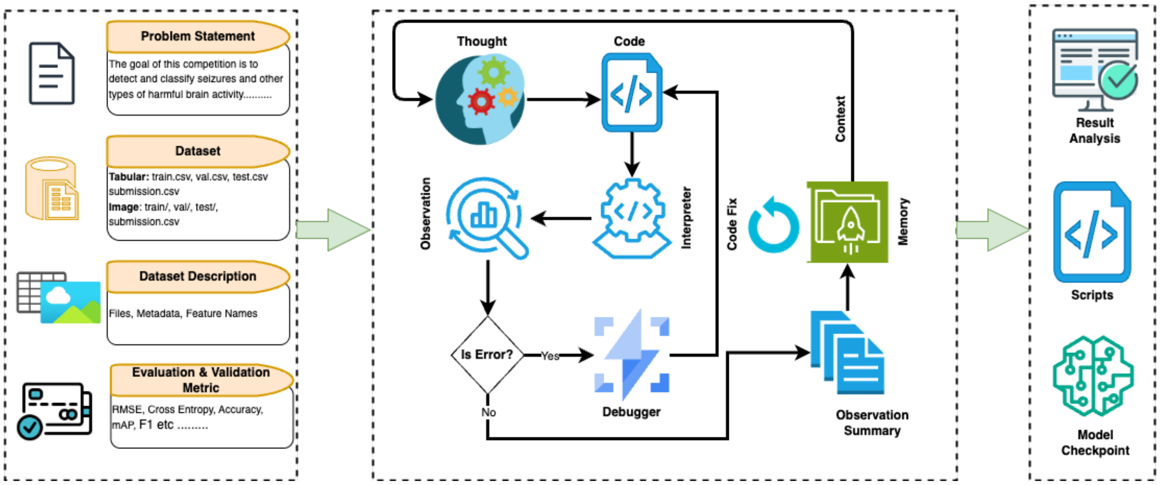

Problem Statement
The goal of this competition is to detect and classify seizures and other types of harmful brain activity.........
Dataset
Tabular: train.csv, val.csv, test.csv submission.csv
Image: train/, val/, test/, submission.csv
Dataset Description
Files, Metadata, Feature Names
Evaluation & Validation Metric
RMSE, Cross Entropy, Accuracy, mAP, F1 etc ....
Thought
Code
Context
Observation
Interpreter
Code Fix
Memory
Is Error?
Yes
No
Debugger
Observation Summary
Result Analysis
Scripts
Model Checkpoint

Figure 1: PiML is an end-to-end framework for autonomous machine learning. Given the problem description, dataset and evaluation metric; framework can iteratively perform EDA, Feature Engineering, modeling and hyper-parameter tuning to obtain the best results.

2.1 Framework Overview

We present an overview of our proposed framework in Figure 1 and Algorithm 1. The workflow starts with a clearly defined Task Description, which—for example, in the case of a Kaggle competition contains details about the competition objective, evaluation criteria, rules, required constraints, and an initial preview of the dataset.

At each iteration, the Task Description and Memory are provided as input to the Main Agent, to generate a Thought and an Action. The Thought represents the agent's reasoning for the current step and the Action corresponds to the generated code. This code is then executed in a code interpreter, to produce an Observation. The Summary Agent analyzes the Observation to generate an Observation Summary. The memory for the next iteration is constructed based on all the previous iterations. If no errors are detected in the observation summary, then the Main Agent goes to the next iteration using the Task Description and Memory. If an error is detected, then the Debug Chain is activated, which will determine the next iteration. This loop continues until one of the two main constraints is met: Time limit or Maximum steps.

2.2 Iterative Refinement Cycle

We formally define a single iteration of our pipeline, which we refer to as Step, as:

$$S_i = (T_i, A_i, O_i)$$

where T_i represents a Thought generated by the Main Agent at S_i, A_i represents an Action (i.e., generated code), and O_i represents an Observation obtained after executing the Action A_i.

Hence, each step S_i represents a single thought-action-observation cycle in our pipeline. The creation of a step is mainly influenced by a fixed Task Description D, which remains unchanged throughout execution, and the Memory of all preceding steps, denoted as M_{i-1}.

We define the agents and functions used in each step. The Main Agent, denoted as $\mathbf{F}_{\text{main}}$, is responsible for generating the Thought and Action based on the input task and memory. The Summary Agent, represented as $\mathbf{F}_{\text{summary}}$, analyzes the observation at each step and produces the Observation Summary denoted as $\hat{O}$. The Code Interpreter, denoted by $\mathbf{f}_{\text{code}}$, executes the action and generates the corresponding observation. The Memory Constructor, represented as $\mathbf{f}_{\text{mem}}$, formulates memory from previous steps for the Main Agent. Finally, the main agent's trajectory, $\boldsymbol{\tau} = \{S_1, S_2, ..., S_{i-1}\}$, represents all the steps up to the previous one. The sequence of operations within each step is as follows:

1. **Memory Construction**: Each iteration begins with memory construction required for the main agent:
$$M_{i-1} = f_{mem}(\tau, L) \quad \text{where } L \text{ is the predefined max token limit.}$$
2. **Thought and Action Generation**: The Main Agent produces a thought and action based on task description (D) and memory (M).
$$(T_i, A_i) = F_{main}(D, M_{i-1})$$
3. **Execution and Observation**: The action (generated code) is executed to produce an Observation.
$$O_i = f_{code}(A_i)$$
4. **Observation Summary**: The observation is analysed and summarized by the Summary Agent to produce Observation Summary.
$$\hat{O}_i = F_{summary}(A_i, O_i)$$
5. **Update Trajectory**: Finally, the current step is added to the agent trajectory.
$$\tau = \tau \cup S_i$$

2.3 Result Generation and Optimization

During pipeline execution, the Main Agent autonomously generates submission files, conditioned on its internal reasoning and the feedback acquired through observations. Each submission file constitutes a complete solution attempt for the specified Task (Kaggle competition). The collection of all such submission files generated throughout pipeline execution forms the final set of results.

Formally, let us define the set of submissions created by our pipeline as:

$$R = \{R_1, R_2, \ldots, R_N\}$$

Algorithm 1 Overall procedure of PiML

```
Require:
    • $D$: Task description ( Kaggle competition)
    • $\bar{T}$: Maximum time limit
    • $TS$: Maximum step limit
    • $DS$: Maximum debug-step limit
    • $M_0$: Initial (empty) memory
    • $R$: Initial (empty) submission set
    • $O_{\text{default}}$: Default observation if LLM fails to produce valid code
Ensure:
    • $R$: Set of submissions $\{R_1, R_2, \ldots, R_N\}$
Initialize $i \leftarrow 0$, $t \leftarrow 0$, $j \leftarrow 0$,$R \leftarrow \emptyset$
while $t < \bar{T}$ and $i < TS$ do
    $(T_i, A_i) \leftarrow \text{MainAgent}(D, M_{i-1})$    ▹ (a) Generate next step
    if isValidCode($A_i$) then
        $O_i \leftarrow \text{ExecuteCode}(A_i)$    ▹ (b) Execute code and get observation
    else
        $O_i \leftarrow O_{\text{default}}$    ▹ Assign default observation if no valid code
    end if
    $\hat{O}_i \leftarrow \text{SummaryAgent}(O_i, A_i)$    ▹ (c) Get observation summary
    if isValid($A_i$) and isError($\hat{O}_i$) then
        $(T_i, A_i, O_i) \leftarrow \text{DebugChain}(T_i, A_i, O_i, DS, D)$    ▹ (d) Debug
        $i \leftarrow i + 1$
    end if
    if SubmissionGenerated then
        $R \leftarrow R \cup R_j$    ▹ (e) Update the submission set
        $j \leftarrow j + 1$
        UpdateSubmissionTracker()
    end if
    $M_i \leftarrow \text{ConstructMemory}(S_1, \ldots, S_i)$    ▹ (f) Construct the memory
    $i \leftarrow i + 1$
    $t \leftarrow t + \text{TimeConsumedThisIteration}()$    ▹ (g) Increment step and time
end while
return $R$
```

Each submission R_j, for $j \in \{1, 2, \ldots, N\}$, is defined as a sequence of executed steps:

$$R_j = \{S_k, S_{k+1}, \ldots, S_i\}, \quad i \leq TS - k + 1$$

where S_i represents the ith step (current step in the pipeline) executed towards submission R_j, and k indicates first step executed towards submission R_j.

Through iterative generation of multiple submissions $(R_1, R_2, \ldots, R_N)$, our pipeline progressively explores diverse solution approaches and systematically leverages intermediate feedback. This iterative refinement mechanism facilitates optimization, maximizing the likelihood of achieving superior outcomes within the predefined limits.

2.4 Memory Construction Dynamics

At each pipeline step S_i, the memory M_i utilized to inform the subsequent step S_{i+1} is constructed systematically from previously executed steps while observing predefined token-length limitations.

Let the cumulative set of recorded historical steps up to current step i be denoted as:

$$S = \{S_1, S_2, \ldots, S_i\}, \quad \text{where} \quad S_i = (T_i, A_i, O_i, \hat{O}_i).$$

Each step S_i within the trajectory consists of a Thought T_i, Action A_i, Observation O_i, and summarized Observation $\hat{O}_i$. The memory M_i at step i is specifically constructed according to one of the following predefined selection strategies (cases), strictly adhering to the token-length constraint L:

Case 1: **Comprehensive Memory**: Emphasizes recent context while retaining older actions:

$$M_i = \{A_k \mid k = 1, \ldots, i-10\} \cup \{(T_j, A_j, \hat{O}_j) \mid j = i-9, \ldots, i-1\} \cup \{(T_i, A_i, O_i\}$$

Case 2: **Reduced Recent Context**: Retains older actions but includes only the immediate current step:

$$M_i = \{A_k \mid k = 1, \ldots, i-1\} \cup \{(T_i, A_i, O_i\}$$

Case 3: **Historical Action Chain**: Represents an abbreviated action history to meet the strictest token constraints, beginning from the earliest possible step m within the token limit:

$$M_i = \{A_k \mid k = m, \ldots, i-1\} \cup \{(T_i, A_i, O_i\}, \quad m \geq 1$$

Case 4: **Current Step Context Only**: This minimal context involves only the current step details:

$$M_i = \{(T_i, A_i, O_i\}$$

Case 5: **Minimal Immediate Memory**: The simplest and most concise form:

$$M_i = \{(A_i, O_i)\}$$

At each memory-construction stage, we sequentially choose the first case from the ordered list above that satisfies token-limit constraints. This systematic approach ensures an optimal and balanced inclusion of historical and recent context, enabling efficient decision-making in subsequent pipeline iterations.

2.5 Debug Chain: Systematic Error Correction

When an error is detected in the observation summary $\hat{O}_i$ of the ith step the debug chain is activated. The debug chain iteratively refines the Action A_i from the step until the error is resolved or the maximum debug depth DS is reached. In the case, where the maximum debug depth is reached and the error is still not resolved, the enriched context obtained from the exploration of the debug chain will help the main agent tackle the error in a more informed way. Debug chain outputs the current step (S_i^e) post debugging as follows:

$$S_i^e = (T_i^e, A_i^e, O_i^e)$$

where T_i^e is the final thought that summarizes all debug iterations, A_i^e is the final action, either the corrected one or the last attempt, and O_i^e is the corresponding observation of the final action.

2.6 Stopping criteria and outcomes

The pipeline terminates based on two primary stopping criteria: reaching the maximum time limit $\bar{T}$ or the maximum number of steps TS. These parameters can be adjusted based on the specific input task. The final outcome of the pipeline is the complete set of submission files R generated during the execution.

---

[0] e notation denotes the Thought, Action and Observation post Debug Chain

3 Experiments

In this section, we evaluate the flexibility and effectiveness of our PiML framework by applying it to a subset of MLE-Bench (Chan et al., 2024). We carefully curate a subset from MLE-Bench spanning across all competition category and available complexity mix. Full details of our dataset selection criteria are provided in Section 3.3

3.1 Experimental Setup

All experiments are conducted using Microsoft Azure's Standard NC24ads A100 v4 virtual machines, each equipped with 24 vCPUs, 220 GiB memory, and a single Nvidia A100 GPU (80GB). Unlike MLE-Bench's original setup, which runs agents on Standard NV36ads A10 v5 instances (36 vCPUs, 440 GiB memory, Nvidia A10 GPU with 24GB). Another key distinction is that we execute two competitions in parallel, where each agent shares the available compute resources, our setup differs due to budget and hardware availability, yet comparable and sufficient to produce results.

Each agent operates within an Ubuntu 20.04 Docker container, preloaded with the dataset, a validation server, and essential Python packages for ML engineering. Agents have a maximum of 24 hours per competition to generate submissions. To ensure fair evaluation, we consider all intermediate submissions made by an agent, rather than only the final submission.This approach allows us to capture the iterative learning process of the agent and assess problem-solving capabilities beyond a single final output. This approach aligns with the methodology of MLE-Bench for reporting baselines, where multiple submissions across different seeds are aggregated to determine the best-performing attempt.

3.2 Baseline

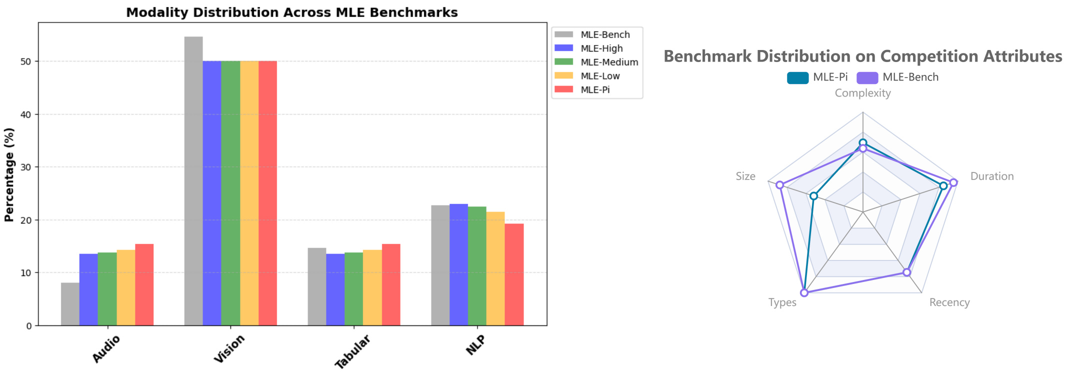
We employ AIDE (Jiang et al., 2025) as our primary baseline, which is the best-performing framework in MLE-bench evaluation results. We use AIDE's default settings, only modifying agent's (agent.code.model) underlying model. For other agent specific parameter refer Appendix A.2. Additionally, we report numbers from ResearchAgent (referred to as "MLAB") from MLAgentBench (Huang et al., 2023), and CodeActAgent (referred to as "OpenHands") from the OpenHands platform (Wang et al., 2024), specifically for runs using GPT-4o. These results are sourced directly from the MLE-Bench paper (Chan et al., 2024), where we filter for the best submission across all available seeds before computing our final evaluation metrics. This further justifies our decision to consider intermediate submissions, ensuring fairer comparisons across multiple runs.

3.3 Dataset

MLE-bench (Chan et al., 2024) is an offline Kaggle competition environment designed to evaluate AI agents on real-world machine learning tasks. Each competition has an associated description, dataset, and grading code. Agents are required to design, build, and train models on GPUs, with submissions graded locally and compared against real-world human attempts via the leaderboards.

MLE-Bench officially splits its tasks into three subsets based on complexity tiers: "Low", "Medium", and "High". However, these predefined splits do not fully capture the overall dataset distribution across modality, competition types, dataset sizes, and complexity variations. Due to budget and resource constraints, we select a diverse subset of 26 competitions that balances representation across these factors while maintaining computational feasibility. We refer to this derived dataset as **MLE-Pi** for simplicity. Refer to Appendix A.1 for complete list and details.

To validate MLE-Pi as a representative subset, we compare its distributional characteristics against the full MLE-Bench. Figure 2 confirms that MLE-Pi preserves key statistical properties of the complete benchmark, making it a reliable proxy for evaluation. Moreover, given its alignment with the overall dataset, insights and trends observed on MLE-Pi extend to the full set of 75 competitions, reinforcing its suitability as a practical and computationally efficient extension of MLE-Bench.



(a) Modality distribution across MLE Benchmarks (b) MLE-Pi across competition statistics

Figure 2: Benchmark distribution on competition attributes and modality, comparing complexity, dataset size, types, recency and duration.

3.4 Results

We evaluate the performance of the PiML framework in Table 1 on the following metrics:

- Valid Submission (%) – Percentage of competitions where the agent generates a valid submission.
- Submissions Above Median (%) – Percentage of competitions where the agent's best submission outperforms the human median level (50th percentile).
- Average Percentile Rank (%) – The mean leaderboard percentile achieved across all competitions.
- Any Medal (%) – Percentage of competitions where the agent earns at least a bronze medal.

We follow Kaggle progression system (Kaggle, 2024) to determine the medals earned by the agent in the competitions following the same convention as MLE-Bench (Chan et al., 2024).

Table 1: Agent Performance on MLE-Pi

Model	Agent Framework	Valid Submission (%)	Submissions Above Median (%)	Average Percentile (%)	Any Medal (%)	GOLD	SILVER	BRONZE
gpt-4o-2024-08-06	AIDE*	80.76	23.08	30.36	19.23	3	1	1
	MLAB*	73.07	7.69	16.89	3.85	0	1	0
	OpenHands*	80.76	19.23	23.63	7.69	0	1	1
	PiML (Ours)	76.92	26.92	33.67	23.08	1	3	2
o3-mini : high	AIDE	92.31	30.77	39.06	23.08	2	2	2
	PiML (Ours)	92.31	34.61	43.75	26.92	4	2	1

PiML proves to be the most effective framework on MLE-Pi. It consistently achieves the highest average percentile across settings—33.67% under gpt-4o[1] OpenAI (2024) and 43.75% under o3-mini[2](OpenAI, 2025c) — outperforming all other approaches and reaching closer to median human performance. It also secures the most gold medals (4) with o3-mini[2], reinforcing its competitive strength. All our experiments can be reproduced using scripts available at Anonymous Repository[3]

We also conducted experiments comparing traditional AutoML systems v/s PiML (LLM Driven Framework). Our objective was to analyze the ability of both frameworks to work within the

---

[1] gpt-4o-2024-08-06

[2] o3-mini-2025-01-31 with reasoning effort - *High*

[3] PiML Anonymous Repository

constraints of a downstream task and ability to generalize to new domains and assess their performance wrt experimentation ability etc. Our findings suggest that PiML's iterative refinement strategy enhances generalisation and provides deeper insights into the model's behaviour across different domains. Refer to Appendix [A.4] for more details on experiments.

3.5 Discussion

In our study, we have observed that the PiML methodology demonstrates several advantages over AIDE (Jiang et al., 2025), particularly in its interactive and human-readable approach. Built on the ReAct framework (Yao et al. (2023b)), PiML enables real-time data interpretation, facilitating early error detection and adaptive decision-making. Its Jupiter-style coding environment enhances transparency by exposing intermediate results, creating an iterative feedback loop essential for dynamic analysis.

In contrast, AIDE(Jiang et al., 2025) follows an iterative refinement process to generate end-to-end solutions. While this ensures systematic modifications, it slows response to immediate data feedback. Its atomic changes maintain rigor but hinder rapid error correction or strategic pivots, making it less efficient in navigating NP-hard search spaces. Additionally, AIDE's(Jiang et al., 2025) greedy selection process may limit creative exploration, especially in dynamic environments. Its inability to persist computations forces full re-execution on error, increasing inefficiency with large datasets.

Moreover, the inherent trade-offs between transparency and automation are evident when comparing the two methodologies. PiML's reliance on intermediate outputs not only enhances interpretability but also promotes a level of creativity and flexibility by allowing for rapid pivots as new observations emerge. PiML truly shines when dealing with large datasets. In such cases, the loading and processing times are substantial, and any error in AIDE(Jiang et al., 2025) necessitates a complete re-execution of all steps because the environment does not persist previous computations. While, PiML enables quick error fixes and the exploration of new methods without the risk of losing prior progress, offering a more human-centric and efficient approach to handling complex data challenges.

Ultimately, the choice between PiML and AIDE(Jiang et al., 2025) is context-dependent. In scenarios where exploratory data analysis and rapid prototyping are required, like in Machine Learning, PiML's interactive, feedback-driven methodology is likely to offer significant advantages. However, for applications that demand robust, fully automated code generation in well-defined settings, the systematic nature of AIDE(Jiang et al., 2025) may be more appropriate despite its potential drawbacks in flexibility and responsiveness.

3.6 Limitations

Reliability on Underlying LLM for ML Code Generation: PiML's performance is significantly dependent on the quality and reliability of the underlying large language model (LLM) used for generating machine learning code. As highlighted in the ML Code Efficiency Report (Appendix A.3), inconsistencies or biases in the LLM may affect the overall code efficiency and correctness.

Offline Mode of Operation: The methodology currently operates in an offline manner, relying entirely on the pre-existing knowledge of the LLM. This lack of real-time or online learning capabilities can limit its adaptability to new data or emerging trends, underscoring the need for an online, continuously updating approach.

Seed Randomness Impact: The initialization randomness can significantly affect the reproducibility and consistency of results. Variations in random seed values may lead to different outcomes, which challenge the reliability and repeatability of experiments conducted using PiML.

Lack of Visual Clues via Plots Understanding: Although incorporating context from vision-language models (VLMs) shows promise, the current framework falls short in effectively integrating visual cues from graphs or other visual data representations. While preliminary experiments indicate that visual context can be meaningful (Appendix A.5), there remains a pressing problem in determining how best to leverage these insights to enhance model performance and interpretability.

These limitations highlight areas where future development could focus on improving the adaptability, reliability, and overall robustness of the PiML methodology.

4 Related Work

4.1 Large Language Models

LLMs with their vast amount of prior knowledge and their ability to reason and perform actions; have opened up a variety of agentic applications, such as autonomous code generation (Le et al. (2022), Singh et al. (2025)) and debugging (Chen et al. (2023), Zhong et al. (2024)), complex decision making in domains like finance, healthcare and patient care (Peng et al. (2023), Busch et al. (2025)), and research automation (Gottweis et al. (2025), Lu et al. (2024)).

4.2 Impact of advancements in LLM Reasoning on ML Tasks

Research in Generative AI domain has been moving from System-1 models (OpenAI (2024), Anthropic (2024)) which rely on fast and intuitive pattern recognition and were majorly relying on their vast knowledge base for solving downstream tasks to System-2 models (OpenAI (2025b), OpenAI (2025c), DeepSeek-AI et al. (2025), Anthropic (2025)) that incorporate reasoning, planning and structured decision making. This shift involves multi-step planning, tool use, search etc, making AI more capable of solving complex problems. These advancements have bolstered the development of end-to-end autonomous machine learning workflows. AutoML requires the system to find the optimal pairing of data preprocessing - EDA, Feature Engineering, Modeling and HPO to get the best results in an efficient manner. Traditional AutoML frameworks (Erickson et al. (2020b), LeDell et al. (2020), Feurer et al. (2015)) rely on predefined algorithms and heuristic strategies to automate machine learning tasks. However, these approaches only give sub-optimal results as they tend to treat the sub-problems independently and lack the holistic overview of the problem, along with iterative experimentation.

4.3 Solving Long Complex Task using Agentic Systems

Building on top of the frameworks which aimed to solve different stages of ML workflow, many works targated building end-to-end agentic autonomous systems capable of solving general multi-turn challenging problems, machine learning challenges and more. OpenHands (Wang et al., 2024) attempted to automate software development tasks of a human developer embedding code interaction, execution and web search into its framework. OpenManus (Liang et al., 2025) the open-source counterpart for (manus.im, 2025) enables automated general AI Agent by leveraging multi-step planning, code execution, tool integration and web search.

Weco AI's AIDE (Jiang et al., 2025) designed a tree based approach for exploring multiple solution paths for a ML problem by introducing atomic changes in an iterative way. AutoKaggle (Li et al., 2024) developed a multi-agent collaborative system of five specialised agents: Reader, Planner, Developer, Reviewer and Summarizer operating across different stages of ML workflow. DS-Agent (Guo et al., 2024) integrates LLMs with case based reasoning (CBR) by leveraging past solutions from Kaggle to generate and refine ML experiment plans. Agent-K (Grosnit et al., 2024) introduced an autonomous data science system using a structured reasoning framework with nested memory processing, enabling continuous improvement and adaptive decision-making. These frameworks highlight the growing trend of employing autonomous agents to enhance diverse workflows, reduce human intervention, and improve the efficiency of complex tasks across multiple domains.

5 Conclusion

We introduce PiML, an iterative refinement framework for solving real-world machine learning tasks. Our framework adopts long term planning and reasoning, adaptive memory and systematic step-by-step debugging to tackle complex problems efficiently. We show performance improvement of PiML over AIDE(Jiang et al., 2025), OpenHands, MLAB on a challenging MLE-Pi Dataset (subset of MLE-Bench dataset). Our findings reveal the importance of adaptive capability towards different domains and the need for context-aware reasoning for solving complex ML tasks.

6 Future Directions

The quest to solve complex problems that evolve over extended periods remains a central driving force in artificial intelligence research. Long-horizon tasks inherently require a sequence of deliberate actions and decisions executed over time to achieve specific objectives. This challenge spans across diverse domains such as software development and scientific research, where initiatives like Claude Code (Anthropic (2025)) and AI Co-Scientist (Gottweis et al. (2025)) have made significant strides.

Innovative systems like Manus AI (manus.im (2025)) highlight the emerging trend towards creating generalist agents capable of autonomously handling a wide range of tasks—from designing websites and analyzing stock market trends to planning travel itineraries. These examples underscore the potential for AI to manage multifaceted projects with minimal human oversight, continuously learning and adapting through trial and error to refine their decision-making processes.

In the realm of Machine Learning and Deep Learning, the need for long-term, iterative learning is paramount. Success in these areas depends on persistent refinement, where each cycle of trial, error, and reasoning paves the way for incremental improvements. The promising results demonstrated by PiML on MLE-Pi illustrate both the efficacy of current methods and the untapped potential for developing even more robust, end-to-end machine learning workflows. Future research could leverage large language models to streamline ML workflows by incorporating continuous learning, where insights from one experiment directly subsequent endeavors. By adopting self-evolution techniques, AI systems could iteratively refine their own architectures and training processes.

References

Anthropic. Introducing claude 3.5 sonnet, 2024. URL `https://www.anthropic.com/news/claude-3-5-sonnet`.

Anthropic. Claude 3.7 sonnet and claude code, 2025. URL `https://www.anthropic.com/news/claude-3-7-sonnet`.

Anthropic. Claude Code, 2025. URL `https://docs.anthropic.com/en/docs/agents-and-tools/claude-code/overview`. Version 1.0.

Felix Busch, Lena Hoffmann, Christopher Rueger, Elon HC van Dijk, Rawen Kader, Esteban Ortiz-Prado, Marcus R Makowski, Luca Saba, Martin Hadamitzky, Jakob Nikolas Kather, et al. Current applications and challenges in large language models for patient care: a systematic review. *Communications Medicine*, 5(1):26, 2025.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Lilian Weng, and Aleksander Mądry. Mle-bench: Evaluating machine learning agents on machine learning engineering. *arXiv preprint*, arXiv:2410.07095, 2024. URL `https://arxiv.org/abs/2410.07095`.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug, 2023. URL `https://arxiv.org/abs/2304.05128`.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025. URL `https://arxiv.org/abs/2501.12948`.

Nick Erickson, Jonas Mueller, Alexander Shirkov, Hang Zhang, Pedro Larroy, Mu Li, and Alexander Smola. Autogluon-tabular: Robust and accurate automl for structured data, 2020a. URL `https://arxiv.org/abs/2003.06505`.

Nick Erickson, Jonas Mueller, Zeren Zhang, Alexander Mao, Alex Smola, et al. Autogluon-tabular: Robust and accurate automl for structured data. *arXiv preprint arXiv:2003.06505*, 2020b.

Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems 28 (NeurIPS 2015)*, pages 2962–2970, 2015. URL `https://proceedings.neurips.cc/paper/2015/file/11d0e6287202fced83f79975ec59a3a6-Paper.pdf`.

Nanxu Gong, Chandan K Reddy, Wangyang Ying, Haifeng Chen, and Yanjie Fu. Evolutionary large language model for automated feature transformation. *arXiv preprint arXiv:2405.16203*, 2024.

Juraj Gottweis, Wei-Hung Weng, Alexander Daryin, Tao Tu, Anil Palepu, Petar Sirkovic, Artiom Myaskovsky, Felix Weissenberger, Keran Rong, Ryutaro Tanno, et al. Towards an ai co-scientist. *arXiv preprint arXiv:2502.18864*, 2025.

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

Siyuan Guo, Cheng Deng, Ying Wen, Hechang Chen, Yi Chang, and Jun Wang. Ds-agent: Automated data science by empowering large language models with case-based reasoning, 2024. URL `https://arxiv.org/abs/2402.17453`.

Noah Hollmann, Samuel Müller, and Frank Hutter. Large language models for automated data science: Introducing caafe for context-aware automated feature engineering. *Advances in Neural Information Processing Systems*, 36:44753–44775, 2023.

Qian Huang, Jian Vora, Percy Liang, and Jure Leskovec. Mlagentbench: Evaluating language agents on machine learning experimentation. *arXiv preprint arXiv:2310.03302*, 2023.

Daniel P Jeong, Zachary C Lipton, and Pradeep Ravikumar. Llm-select: Feature selection with large language models. *arXiv preprint arXiv:2407.02694*, 2024.

Zhengyao Jiang, Dominik Schmidt, Dhruv Srikanth, Dixing Xu, Ian Kaplan, Deniss Jacenko, and Yuxiang Wu. Aide: Ai-driven exploration in the space of code. *arXiv preprint arXiv:2502.13138*, 2025.

Kaggle. Kaggle progression system, 2024. URL `https://www.kaggle.com/progression`. Accessed: 2024-03-27.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning, 2022. URL `https://arxiv.org/abs/2207.01780`.

Erin LeDell, Sebastian Poirier, et al. H2o automl: Scalable automatic machine learning. *19th Python in Science Conference*, pages 111–120, 2020.

Dawei Li, Zhen Tan, and Huan Liu. Exploring large language models for feature selection: A data-centric perspective. *ACM SIGKDD Explorations Newsletter*, 26(2):44–53, 2025.

Ziming Li, Qianbo Zang, David Ma, Jiawei Guo, Tuney Zheng, Minghao Liu, Xinyao Niu, Yue Wang, Jian Yang, Jiaheng Liu, et al. Autokaggle: A multi-agent framework for autonomous data science competitions. *arXiv preprint arXiv:2410.20424*, 2024.

Xinbin Liang, Jinyu Xiang, Zhaoyang Yu, Jiayi Zhang, and Sirui Hong. Openmanus: An open-source framework for building general ai agents. `https://github.com/mannaandpoem/OpenManus`, 2025.

Siyi Liu, Chen Gao, and Yong Li. Large language model agent for hyper-parameter optimization, 2025. URL `https://arxiv.org/abs/2402.01881`.

Chris Lu, Cong Lu, Robert Tjarko Lange, Jakob Foerster, Jeff Clune, and David Ha. The ai scientist: Towards fully automated open-ended scientific discovery, 2024. URL `https://arxiv.org/abs/2408.06292`.

Simon Malberg, Edoardo Mosca, and Georg Groh. Felix: Automatic and interpretable feature engineering using llms. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 230–246. Springer, 2024.

manus.im. Manus | The general AI agent, March 2025. URL `https://manus.im/`.

OpenAI. Gpt-4o system card, 2024. URL `https://openai.com/index/gpt-4o-system-card/`.

OpenAI. Gpt-4.5 system card, 2025a. URL `https://openai.com/index/gpt-4-5-system-card/`.

OpenAI. Openai o1 model documentation, 2025b. URL `https://platform.openai.com/docs/models/o1`.

OpenAI. Openai o3-mini system card, 2025c. URL `https://openai.com/index/o3-mini-system-card/`.

Cheng Peng, Xi Yang, Aokun Chen, Kaleb E Smith, Nima PourNejatian, Anthony B Costa, Cheryl Martin, Mona G Flores, Ying Zhang, Tanja Magoc, et al. A study of generative large language model for medical research and healthcare. *NPJ digital medicine*, 6(1):210, 2023.

Oleksandr Shchur, Caner Turkmen, Nick Erickson, Huibin Shen, Alexander Shirkov, Tony Hu, and Yuyang Wang. Autogluon-timeseries: Automl for probabilistic time series forecasting, 2023. URL `https://arxiv.org/abs/2308.05566`.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yue Ting Zhuang. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. *ArXiv*, abs/2303.17580, 2023. URL `https://api.semanticscholar.org/CorpusID:257833781`.

Kunal Singh, Ankan Biswas, Sayandeep Bhowmick, Pradeep Moturi, and Siva Kishore Gollapalli. Sbsc: Step-by-step coding for improving mathematical olympiad performance. *arXiv preprint arXiv:2502.16666*, 2025.

Zhiqiang Tang, Haoyang Fang, Su Zhou, Taojiannan Yang, Zihan Zhong, Tony Hu, Katrin Kirchhoff, and George Karypis. Autogluon-multimodal (automm): Supercharging multimodal automl with foundation models, 2024. URL `https://arxiv.org/abs/2404.16233`.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*, 2024.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023. URL `https://arxiv.org/abs/2201.11903`.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models, 2023a. URL `https://arxiv.org/abs/2305.10601`.

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

Lei Zhang, Yuge Zhang, Kan Ren, Dongsheng Li, and Yuqing Yang. Mlcopilot: Unleashing the power of large language models in solving machine learning tasks, 2024a. URL `https://arxiv.org/abs/2304.14979`.

Shujian Zhang, Chengyue Gong, Lemeng Wu, Xingchao Liu, and Mi Zhou. Automl-gpt: Automatic machine learning with gpt. *ArXiv*, abs/2305.02499, 2023. URL `https://api.semanticscholar.org/CorpusID:258480269`.

Xinhao Zhang, Jinghan Zhang, Banafsheh Rekabdar, Yuanchun Zhou, Pengfei Wang, and Kunpeng Liu. Dynamic and adaptive feature generation with llm. *arXiv preprint arXiv:2406.03505*, 2024b.

Li Zhong, Zilong Wang, and Jingbo Shang. Debug like a human: A large language model debugger via verifying runtime execution step-by-step, 2024. URL `https://arxiv.org/abs/2402.16906`.

Marc-André Zöller and Marco F. Huber. Benchmark and survey of automated machine learning frameworks. *Journal of Artificial Intelligence Research*, 70:409–472, 2021. doi: 10.1613/jair.1.11854. URL `https://arxiv.org/abs/1904.12054`.

A Appendix and Supplemental Material

A.1 Dataset details: MLE-Pi

MLE-Pi is a curated collection of 26 competitions, sampled from MLE-Bench's original set of 75. This subset is carefully constructed to cover all 15 competition categories while incorporating every available complexity level—Low, Medium, and High. The result is a balanced yet computationally efficient proxy for MLE-Bench. Table 2 lists the selected competitions for reference.

Table 2: MLE-Pi Dataset Details

Competition	**Category**	**Size (GB)**	**Complexity**
the-icml-2013-whale-challenge-right-whale-redux	Audio Classification	0.29314	Low
tensorflow-speech-recognition-challenge	Audio Classification	3.76	Medium
ventilator-pressure-prediction	Forecasting	0.7	Medium
histopathologic-cancer-detection	Image (Other)	7.76	Low
petfinder-pawpularity-score	Image (Other)	1.04	Medium
rsna-miccai-brain-tumor-radiogenomic-classification	Image (Other)	135.85	High
leaf-classification	Image Classification	0.036	Low
statoil-iceberg-classifier-challenge	Image Classification	0.3021	Medium
hms-harmful-brain-activity-classification	Image Classification	26.4	High
tgs-salt-identification-challenge	Image Segmentation	0.5	Medium
3d-object-detection-for-autonomous-vehicles	Image Segmentation	125.79	High
denoising-dirty-documents	Image to Image	0.06	Low
vesuvius-challenge-ink-detection	Image to Image	37.02	High
bms-molecular-translation	Image to Text	8.87	High
siim-covid19-detection	Object Detection	128.51	High
text-normalization-challenge-english-language	Sequence to Sequence	0.01	Low
seti-breakthrough-listen	Signal Processing	156.02	Medium
predict-volcanic-eruptions-ingv-oe	Signal Processing	31.25	High
nomad2018-predict-transparent-conductors	Tabular	0.00624	Low
champs-scalar-coupling	Tabular	1.22	Medium
stanford-covid-vaccine	Tabular	2.68	High
us-patent-phrase-to-phrase-matching	Text (Other)	0.00214	Medium

spooky-author-identification	Text Classification	0.0019	Low
tweet-sentiment-extraction	Text Classification	0.003	Medium
google-quest-challenge	Training LLMs	0.015	Medium
nfl-player-contact-detection	Video Classification	5.01	High

MLE-Bench also provides its own subsets, but they are exclusively based on complexity levels ("Low," "Medium," "High") for ease of evaluation. As shown in Figure 2(a), these subsets exhibit similar modality distributions. However, Figure 3 shows MLE-Pi achieves better comprehensive proportional representation by categories compared to predefined subsets. This makes MLE-Pi an ideal stand-in for the full MLE-Bench, particularly for testing, experimentation, and resource constrained scenarios, with its results being clearer extension in performance to actual benchmark.

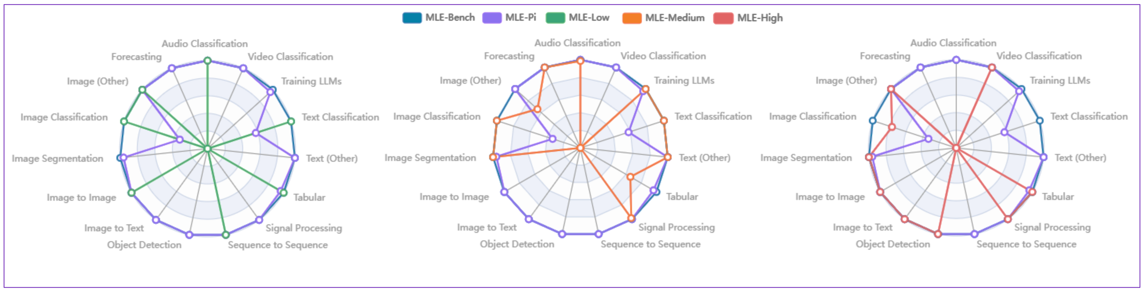


Figure 3: MLE data splits, with proportional scaling for fair comparison to MLE-Bench

The motivation behind this split is the significant resource demands of running MLE-Bench in its entirety. A single experiment run, with each competition allotted 24 hours, requires a total of 1,800 GPU hours (24 hours × 75 competitions). Beyond just compute time, the benchmark also incurs substantial infrastructure, memory, and overall system overhead. Given these issues, MLE-Pi provides a practical yet representative alternative, making benchmarking more accessible without sacrificing diversity or complexity.

A.2 Agent Settings

Table 3 details the hyperparameters for each of our tested scaffolds:

Table 3: Scaffold hyperparameters. `$TARGET_MODEL` is the model being evaluated.

AIDE	
Parameter	**Value**
agent.code.model	`$TARGET_MODEL`
agent.code.reasoning_effort	high
agent.feedback.model	`$TARGET_MODEL`
agent.feedback.reasoning_effort	high
agent.steps	500
agent.search.max_debug_depth	4
agent.search.debug_prob	1
agent.time_limit	86400
exec.timeout	32400

PiML

Parameter	Value
agent.steps	600
agent.llm.model	`$TARGET_MODEL`
agent.llm.temperature	0.5
agent.llm.reasoning_effort	high
agent.debug_steps	10
agent.time_limit	86400
exec.timeout	32400

A.3 Understanding and comparing the Coding Efficiency of LLMs

This section tries to understand and compare the coding efficiency of different Large Language Models for Machine learning problems.

A.3.1 Dataset and LLMs Selection. For this experiment, we use the MLE-Pi dataset, as defined in Appendix A . This dataset provides a diverse collection of Kaggle Competitions, ensuring a balanced representation of both complexities and modalities.

The LLMs selected for our experimentation are gpt-4o (OpenAI, 2024), gpt-4.5-preview (OpenAI, 2025a) , o1(OpenAI, 2025b), o3-mini (medium reasoning effort) (OpenAI, 2025c), o3-mini (high reasoning effort) (OpenAI, 2025c), and deepseek-r1-distill-qwen-32B (DeepSeek-AI et al., 2025)

A.3.2 Coding Efficiency metric. From our experiment logs, we observed that the code generated by LLMs for Machine Learning problems, like Kaggle competitions, often fails to utilize the available resources effectively. In several cases, when GPUs were available and explicitly mentioned in the context, the LLM failed to utilize them in its generated code. Additionally, in some instances, the LLM selected suboptimal models for the given modality, such as choosing scikit-learn models for an image competition. To quantify these inefficiencies, we propose the MLCES (ML Code Efficiency Score) metric.

MLCES metric: The MLCES measures how effectively a machine learning solution generated by LLM utilizes computational resources and selects appropriate models. It evaluates two key factors: GPU usage (G) and model architecture quality (M).

If a task requires a GPU (e.g., image or audio processing), the score assigns:

$$G = \begin{cases} 1, & \text{if the code correctly utilizes a GPU,} \\ 0, & \text{if the GPU is ignored despite being available and required.} \end{cases}$$

For model selection:

$$M = \begin{cases} 1, & \text{if the code employs a competitive model architecture for the task,} \\ 0, & \text{if the model choice is suboptimal (e.g., using scikit-learn for image processing).} \end{cases}$$

The raw score (S) is calculated as follows:

- **For GPU-dependent tasks (e.g., image, audio, GPU-intensive NLP)**:
$$S = G + M \quad \text{(possible values: 0, 1, or 2)}$$
- **For non-GPU tasks (e.g., tabular data)**:
$$S = 2 \times M \quad \text{(possible values: 0 or 2)}$$

To ensure consistency across tasks, the final MLCES is normalized:

$$\text{MLCES} = \frac{S}{2} \quad \text{(yielding a value between 0 and 1)}$$

Interpretation of the score:

- 0.0 ⟶ Neither GPU utilization nor appropriate model selection was applied.
- 0.5 ⟶ Either GPU usage or model selection was correct, but not both.
- 1.0 ⟶ The solution efficiently utilizes the GPU (if required) and selects a competitive model.

A.3.3 **Experimental Setup**. To evaluate and compare the performance of the LLMs, we used a repeated sampling strategy. For each pair of competition and LLM pair, we conducted 50 independent sampling trials. Within each trial, we randomly selected 12 candidate solutions generated by the respective LLM. We calculated the MLCES metric for each solution using a separate gpt-4o (OpenAI, 2024) model. For every trial, we computed the average MLCES score across the 12 sampled solutions and the mean performance per LLM for each competition by averaging these scores over the 50 trials. Finally, to summarize and compare overall performance across competitions, we aggregated these competition level means to obtain a final overall mean and corresponding standard deviation for each LLM.

A.3.4 **Results and Analysis**. From Table 4 and Figure 4a, we observe that most LLMs score below 0.5, with gpt-4.5-preview (OpenAI, 2025a) performing slightly better at 0.56. Reasoning models, in general, perform worse, with o1 (OpenAI, 2025b) being significantly low at 0.24. The only exception is o3-mini-high (OpenAI, 2025c), which scores 0.50, slightly outperforming gpt-4o (OpenAI, 2024) at 0.49. deepseek-r1-distill-qwen-32B (DeepSeek-AI et al., 2025) is comparable to o3-mini-medium (OpenAI, 2025c) at 0.44, though both still score lower than the GPT models.

This consistent low performance across all the top LLMs indicates that further research is needed on ML problem-specific optimization within LLMs. The advancements in reasoning models do not necessarily translate to improved performance in this domain, suggesting that their effectiveness may be problem-specific and limited. However, further investigation is needed before drawing any definitive conclusions.

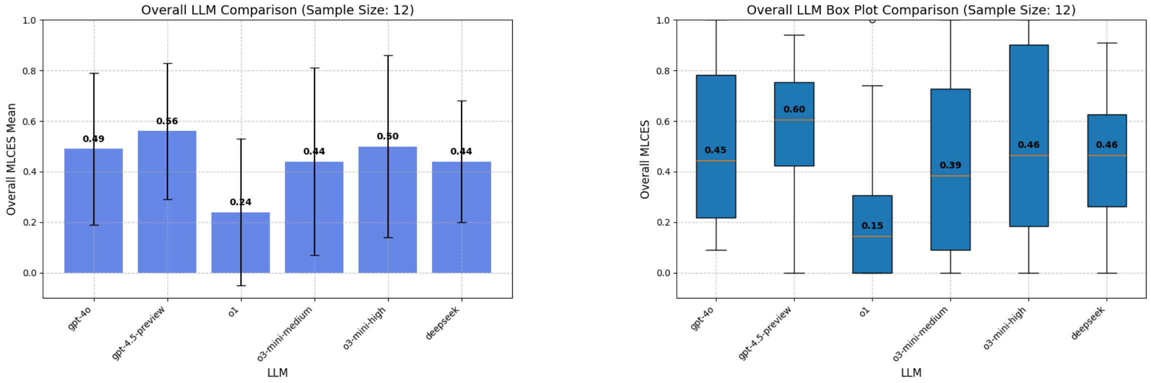
Table 4: Comparison of Overall Machine Learning Coding Efficiency Scores (MLCES) for Various LLMs on the MLE-Pi Dataset

Model	**MLCES** (mean ± std)		
	NCS=4	NCS=8	NCS=12
gpt-4o	0.50 ± 0.30	0.49 ± 0.30	0.50 ± 0.30
gpt-4.5-preview	0.55 ± 0.27	0.55 ± 0.26	0.55 ± 0.27
o1	0.24 ± 0.29	0.24 ± 0.29	0.24 ± 0.29
o3-mini-medium	0.44 ± 0.37	0.44 ± 0.37	0.44 ± 0.37
o3-mini-high	0.50 ± 0.37	0.50 ± 0.36	0.49 ± 0.37
deepseek-r1-distill-qwen-32B	0.45 ± 0.26	0.44 ± 0.24	0.44 ± 0.24

NCS (Number of Candidate Solutions) indicates how many distinct LLM outputs were generated per trial.

A.4 Comparative Analysis of PiML v/s AutoGluon-Tabular

The main objective of these experiments was to understand the capabilities and generalisation abilities of AutoML frameworks and put up a side by side comparison with PiML wrt various



(a) Bar plot comparing the MLCES metric across different LLMs on the MLE-Pi dataset.

(b) Box plot comparing the MLCES metric across different LLMs on the MLE-Pi dataset.

Figure 4: ML coding efficiency of different LLMs

aspects such as interpretability of features, context-aware choice of models or HPO techniques, ability of the framework to work within constraints.

To narrow down the scope of the experiment, we choose AutoGluon-Tabular (Erickson et al. (2020b)) as a reference AutoML framework owing to its popularity in the community and SoTA performance across different frameworks. Further, we select 4 problem statements from the MLE-Pi dataset [Table-2] with Tabular datatype and test them on the AutoGluon-Tabular framework. Our findings [Table-5] suggest PiML, due to its contextual awareness and adaptability towards the new domain, performs far better than the AutoGluon-Tabular framework in 3 out of 4 problem statements. In a specific problem classified as "low" in complexity and does not require extensive exploratory data analysis (EDA), AutoGluon-Tabular performs well. However, PiML achieves comparable performance.

Table 5: Agent Performance on MLE-Pi (Tabular) compared to AutoGluon

Model	Framework	Average Percentile (%)	Number of Medals	GOLD	SILVER	BRONZE
-	AutoGluon-Tabular	25.922	1	0	1	0
gpt-4o-2024-08-06	**PiML (Ours)**	29.064	1	0	1	0
o3-mini : high	**PiML (Ours)**	56.861	2	1	1	0

A.5 Integrating Visual Clues from Plots for Downstream EDA Analysis

Visual analysis is very crucial for obtaining valuable insights from data. It helps in enhansing interpretability of results and improving decision-making. The main objective of this experiment is to understand the impact of visual understanding on key stages of ML Workflow particularly EDA and pre-processing.

For this, we sampled 2 ML problem statements from the MLE-Pi (Appendix-A.1) dataset - champs-scalar-computing and stanford-covid-vaccine. We designed 2 different scenarios - one where we instruct the model to avoid plotting any visualisations and the other where instructions are to visualise the plots wherever necessary. (Sub-section A.5.1). We used OpenAI-o1 (OpenAI (2025b)) for our experiments with `reasoning_effort` set to `meddium` and `max_completion_tokens` to `2048`.

A.5.1 Prompts for EDA.

Prompt for EDA code generation

```
You are an EDA agent assisting the main agent in solving a machine learning
    problem. Your task is to perform exploratory data analysis (EDA) on the given
    dataset by generating Python code.

Dataset Details:
- Dataset Folder Path: ’{dataset_folder_path}’
- Kaggle Competition: ‘{kaggle_competition}‘
- Dataset Description: ‘{dataset_description}‘
- Domain Info: ’{domain_info}’
- EDA directions: ’{eda_directions}’

# <Only if visualisations not required add below text>
Avoid visualization commands - use statistical summaries instead
```

Prompt for getting observations from code and output

```
You are an EDA analysis agent tasked with interpreting the results of an
    exploratory data analysis (EDA) process. Your goal is to extract key
    observations and suggest potential future explorations based on the provided
    details.

Provided Information:
- Kaggle Competition: ‘{kaggle_competition}‘
- Domain Information: ‘{domain_info}‘
- Dataset Description: ‘{dataset_description}‘
- EDA Code: ‘{code}‘
- EDA Results: *(Provided below the prompt)*

Guidelines for Analysis:
1. Key Observations:
   - Summarize meaningful insights derived from the EDA results.
   - Focus on trends, patterns, anomalies, correlations, and distributions.
   - Avoid speculation - observe strictly based on the results.

2. Potential Future Explorations:
   - Suggest logical next steps based on the EDA findings.
   - Include further statistical analysis, feature engineering ideas, or
    additional data collection strategies.
   - Consider possible domain-specific explorations that could enhance model
    performance.

Output format:

- Observations (from code and results)
- Potential Future Explorations

Ensure that your analysis is concise, structured, and data-driven.
```

A.5.2 Analysing Observations.

The EDA of champs-scalar-coupling with images have plots like scc vs distance, scc vs muliken charge, count of coupling types etc offering a more structural breakdown and domain specific analysis whereas, EDA with statistical analysis is more focused on dealing with aggregated feature like mean reactivity. In essence, the statistical analysis offers high level perspective, but lacks structural representation and interpretability resulting in inability to obtain insights into the problem. As it operates on aggregated features, it has a characteristic of looking at a broader perspective. Combining the broader perspective to EDA with image is expected to improve performance. The similar characteristic is observed in stanford-covid-vaccine contest too.

Below, we have provided observations(with image and without image). We have also presented the comparison between observations.

Comparison between observations (champs-scalar-computing)

```
Note:
1. Observation-1 (w/ Image)
2. Observation-2 (w/o Image)

Assessment Summary
1. Relevance to Competition:
   - Observation 1 directly ties chemical and physical factors (bond distance,
    Fermi Contact, Mulliken charges) to scalar coupling, aligning closely with NMR
    theory and prediction goals.
   - Observation 2 provides useful dataset insights but is more focused on broad
    statistical summaries rather than deep feature relationships.
2. Scientific Soundness:
   - Observation 1 aligns well with established NMR and quantum chemistry
    knowledge, particularly the role of Fermi Contact and distancecoupling trends.
   - Observation 2 correctly summarizes dataset properties but lacks deeper
    chemical interpretation.
3. Actionability:
   - Observation 1 suggests direct feature engineering strategies: bond angles,
    torsion angles, per-type modeling, and emphasizing Fermi Contact.
   - Observation 2 suggests refining Mulliken charge features and handling data
    granularity, but with less domain specificity.
4. Domain Alignment:
   - Observation 1 maps well to standard NMR principles, explaining why trends
    exist.
   - Observation 2 reaffirms known statistical properties but lacks detailed
    physical reasoning.
5. Model Impact:
   - Observation 1 offers a clear roadmap for domain-driven features that could
    significantly improve predictions.
   - Observation 2 provides useful but broader modeling suggestions, such as
    handling outliers and per-type separation.
Overall Assessment
- Observation 1 is the stronger foundation for modeling due to its deep physical
    insights and feature engineering strategies.
- Merging key ideas from Observation 2 (e.g., refining Mulliken charge features,
    outlier handling) with Observation 1s domain-driven approach would create a
    well-rounded strategy.
```

Observations with Images (champs-scalar-computing)

```
Scalar Coupling Distribution
- Multimodal distribution with a large cluster near zero and a peak at 90100 Hz
    (mostly 1JHC).
- 1J couplings (1JHC: ~95 Hz, 1JHN: ~48 Hz) are higher than 2J/3J, which cluster
    near zero or negative values.

Contribution Breakdown
- Fermi Contact (fc) dominates scalar coupling (correlation ~0.9999).
- The total coupling is precisely the sum of fc, sd, pso, and dso, though fc is
    the primary driver.

Geometry & Coupling
- 1JHC occurs at short distances (~1 Å), while 2J/3J show larger separations and
    lower couplings.
- Clear distancecoupling relationship: shorter bonds  higher (positive) couplings;
    longer bonds  smaller/negative.

Mulliken Charges
- Distinct charge clusters influence coupling values, indicating local electron
    density effects.
- Higher Mulliken charges (~0.10.2) often associate with stronger 1J couplings
    (~100 Hz).

Molecular Properties
- Potential energy (~400 kcal/mol) and dipole moment (~23 Debye) exhibit global
    trends but weak direct correlation with scalar coupling.
```

Observations without Images (champs-scalar-computing)

```
Dataset Overview
- Train: 4,191,263 rows, Test: 467,813 rows.
- Each row represents an atom pair within a molecule with a scalar coupling
    constant (train) or a placeholder (test).

- Scalar Coupling Constant
- Ranges from -44.76 to 207.71 (Mean: 15.92, Std: 34.94).
Skewed distribution: 50% below 2.28, but upper quartile extends beyond 7.39.

Coupling Types
- Most frequent: 3JHC (1.36M), 2JHC (1.03M); Least: 1JHN (39K).
- 1JHC has the highest mean (94.97), 2JHH is negative (-10.28).

Scalar Coupling Contributions
- Dominated by Fermi Contact (fc) (correlation ~0.9999).
- Diamagnetic spin-orbit (dso) shows moderate correlation (~0.56).
- Spin-dipolar (sd) and Paramagnetic spin-orbit (pso) have weaker effects.

Distance & Coupling
- Mean interatomic distance: ~2.36 Å.
- 1J couplings occur at shorter distances (~1.011.09 Å), 3J at >3.0 Å.
```

```
Quantum Features
- Potential energy (-0.017 correlation) and Mulliken charge (~0 correlation) show
    little direct impact, suggesting the need for localized descriptors.
```

As the impact of image may not be clear due to many visualisations in the result; below we have given one of the images and key observations from that image by performing EDA of champs-scalar-coupling data to understand the importance of image analysis in Machine Learning. The prompt for getting observations for image has kaggle competition description, domain information and dataset description. The observation output first understands the details in plot and information in prompt, and then dives into the observations needed for modeling. We present only the observations below.

Key observations from Image (champs-scalar-computing)

```
Key Insights from distance vs scc(scalar coupling constant) scatterplot
Implications for Modeling
- Distance is clearly an important explanatory variable; however, distance alone
    does not capture all of the variability (especially for 2J vs. 3J).
- Angles/torsions can distinguish cases where two atoms might be 3 bonds apart yet
    still yield large or small couplings depending on the conformation.
- Quantum-chemical properties (Mulliken charge, shielding tensors, partial bond
    orders) add nuance to predict subtle variations.

Cluster Interpretation by coupling type
- 1J (direct bonds): Shortest distances, highest J-coupling (~40200 Hz).
- 2J (2 bonds apart): Intermediate distances (~1.82.2 Å), moderate coupling (~530
    Hz).
- 3J (3 bonds apart): Longer distances (>2.4 Å), smallest coupling but
    angle-dependent.

Summary
- For the Kaggle task, building features around distances, angles, Mulliken
    charges, and shielding tensors will likely improve predictions.
- Coupling types naturally cluster by distance ranges and magnitude of
    J-couplingthis suggests separate modeling approaches (or a single model with
    coupling-type-specific features) could be beneficial.
```

The prompts, observations in this section are LLM generated and summarized for brevity.

A.6 Prompts for our Agents

In this section,we provide all the prompts used in our pipeline.

A.6.1 Main Agent. This subsection provides the ReAct prompt used for our Main Agent, which generates the Thought and Action of a step.

React Prompt for Main Agent

```
Answer the following questions as best you can.
You have access to the following tools:
1. {tool_description}
```

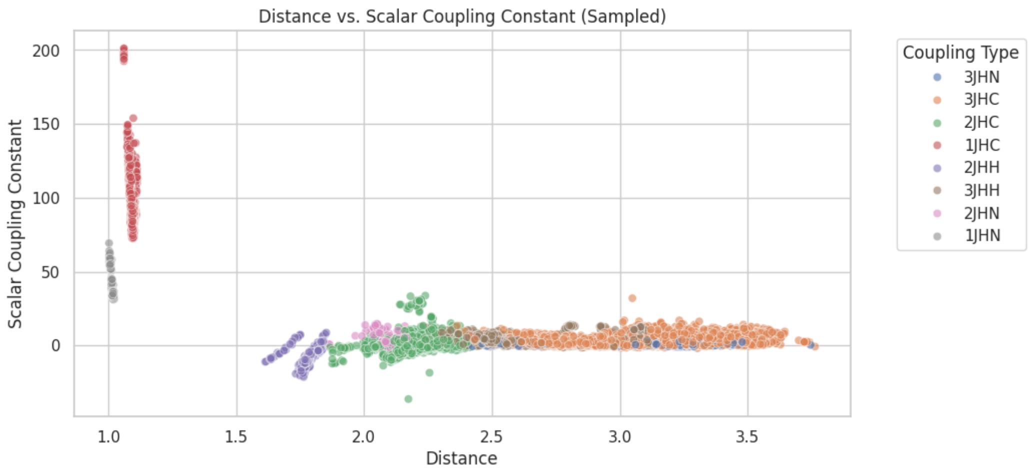


Figure 5: A scatterplot from EDA of champs-scalar-coupling dataset for understanding the visual understanding capabilities of LLMs. There are multiple classes in the image with different colors. The model is able to identify classes and give detailed analysis

```
{instruction}

Your task is to create an initial baseline validation metric and generate a
    corresponding submission_{{validation_metric.csv}} file. Then, continuously
    improve the validation metric and produce updated
    submission_{{validation_metric}}.csv files for each improvement.

Always respond in this format exactly:
Thought: What you are currently doing, what actions to perform and why
Action: the action to take, should be exactly one name of the tool from this list
    - [{tool_names}]
Action Input:
‘‘‘python
[the input to the action]
‘‘‘
Observation: the result of the action
... (this Thought/Action/Action Input/Observation should continue repeating
    indefinitely until instructed to stop)

TASK INSTRUCTIONS:

1. Dataset Management
   - Use only the provided dataset; synthetic datasets are strictly prohibited.
    Sample datasets may be used for quick validation but must be reverted to the
    original dataset afterward
   - When resource constraints prevent using the entire dataset for training, use
    a portion of the original dataset. Always ensure predictions are made on the
    entire test dataset
   - Verify dataset correctness before any processing
   - Use actual target variables from the data. Never use synthetic target
    variables
   - Implement efficient data loading using generators or iterators
   - Apply appropriate batch sizes and data types for memory efficiency

2. File and Directory Structure
   - Write all output files to "./" (current directory)
   - Organize input data in appropriate directory structure based on modality
   - Create proper train/validation splits
   - Maintain checkpoint registry in current directory
   - Save checkpoints with clear timestamps and metrics

3. Framework and Processing
   - Choose an appropriate ML framework for the task, with a preference for
    PyTorch when equally suitable
   - Implement efficient data loading mechanisms (e.g., PyTorch DataLoader,
    TensorFlow tf.data)
   - Process different data modalities appropriately:
     * Images: Handle different formats, sizes, and channels
     * Text: Process different languages, encodings, and lengths
     * Tabular: Handle different datatypes, missing values, and categorical
    variables
     * Sequential: Process variable lengths and temporal dependencies
     * Audio: Handle different sampling rates, durations, and formats
   - Use appropriate libraries for data loading based on modality (e.g.,
    PIL/OpenCV for images, transformers for text, librosa for audio)
   - Avoid visualization commands - use statistical summaries instead

4. Data Analysis and Preprocessing
   - Conduct appropriate exploratory data analysis based on the dataset
    characteristics and modality
   - Consider relevant properties that might impact model performance:
     * Images: Resolution, channels, aspect ratios
     * Text: Length, vocabulary, language characteristics
     * Tabular: Feature distributions, correlations, cardinality
     * Sequential: Sequence lengths, temporal patterns
     * Audio: Duration, frequency characteristics, noise levels
   - Design and implement preprocessing steps specific to the data modality
   - Apply appropriate augmentation techniques where beneficial
   - Adapt the preprocessing pipeline based on initial analysis findings

5. Validation Metric and Iteration:
   - Every time the validation metric is checked:
     - Create a file named ‘submission_metric.csv‘ containing predictions on the
    test data, where ‘metric‘ is the current validation score
     - Even if the metric shows no improvement, still create the corresponding
    submission_metric.csv file
   - After each improvement, continue iterating by exploring new strategies (e.g.,
    feature engineering, advanced models) until optimal results are achieved

6. Checkpoint Management
   - Before loading any checkpoint, verify its existence
   - Load latest checkpoint only if it exists when resuming operations
   - Save new checkpoints after significant operations or improvements
```

```
7. Stopping Condition
   DO NOT STOP processing until one of these explicit conditions is met:
   - You receive a direct "stop" instruction
   - You reach the specified time limit
   - You encounter an unrecoverable error
   Even after achieving good results, continue iterating and improving unless a
    stop condition is met.

8. Resource Management
   - Implement GPU memory cleanup
   - Clear cache between training runs
   - Monitor memory usage and leaks
   - Use appropriate data types to minimize memory consumption
   - Stop and reset approach if persistent errors occur
{extra_instructions}
MAKE SURE YOU FOLLOW THE INSTRUCTIONS WHILE EXECUTION.

{agent_scratchpad}
```

A.6.2 Summary Agent. This subsection provides the prompt and json Schema used for the summary agent.

Prompt for Summary Agent

```
You are a helpful assistant. You will be given a Python code block and its
    corresponding execution output. Your task is to summarize the execution output
    in the specified JSON format.

## Code block:
{code_block}

## Execution output:
{execution_output}
```

Json Schema for Summary Agent

```
{
    "type": "object",
    "properties": {
        "is_bug": {
            "type": "boolean",
            "description": "true if the execution output shows that the execution
    failed or has some bug, otherwise false.",
        },
        "has_csv_submission": {
            "type": "boolean",
            "description": "true if a submission file in the format
    ‘submission_metric.csv‘ is created, otherwise false",
        },
        "submission_file_name_list": {
            "type": "array",
            "items": {"type": "string"},
            "description": "List of submission file names if created; an empty
    list otherwise.",
        },
        "summary": {
            "type": "string",
            "description": (
                "Provide a concise overview of the execution output (2-3
    sentences). "
                "Highlight any key metrics, parameters, or events, such as
    performance scores, "
                "hyperparameter values, or significant observations from the
    execution. "
                "If there are errors or failures "
                "mention them explicitly. This summary should act as a standalone
    description of the output."
            ),
        },
    },
    "required": [
        "is_bug",
        "has_csv_submission",
        "submission_file_name_list",
        "summary",
    ],
}
```

A.6.3 Debug Chain. This subsection presents the prompts used in the debug chain, which consists of two main components:

1. Debug Agent – Refines the action iteratively to resolve the error.
2. Integration – Summarizes the entire debug chain to create the final output thought.

Debug Agent. The prompt and json schema for Debug Agent

Prompt for Debug Agent

```
You are an AI assistant tasked with debugging and correcting the error that
    occurred in the latest code cell of a Jupyter notebook.

You will be provided with the following information:
1. **Main Code History**: A list of code cells executed in the notebook, in the
    order of execution. Each code cell is separated by '# %%'.
2. **Data Preview**: A preview of the data (e.g., a subset of rows or a
    description of the data) used in the current notebook. This helps to
    understand potential data-related issues.
3. **Debugging History**: A list of previous debugging attempts, including errors
    encountered from previous cells. This history helps identify whether the error
    is recurring or if progress is being made.
4. **Current Code**: The latest code cell that raised an error. This is the code
    that needs to be debugged and corrected.
```

```
5. **Current Error**: The latest error message or traceback. This provides context
    on what went wrong and helps identify the specific issue.

## Main Code History
{main_history}

## Data Preview
{data_preview}

## Debugging History
{debug_history}

## Current Code
{current_code}

## Current Error
{current_error}

---

## Based on the information above, please provide the following:

1. **reflection**: A detailed analysis of the error.
   - Identify the root cause.
   - Explain why the error occurred.
   - Include any patterns or trends observed in previous debugging attempts that
    may help explain the issue.

2. **corrected_code**: Provide the corrected Python code cell that should be
    executed next.
   - **Strictly** provide only the Python code.
   - Make sure the code resolves the identified error, fixing the root cause.

3. **is_persistent_error**: Indicate whether the error is recurring.
   - If this error has occurred multiple times based on the ‘debugging history‘,
    set this value to **True**.
   - If this error is isolated to the current execution or is a one-time issue,
    set this value to **False**.

Give your output in the specified json format.
```

Json Schema for Debug Agent

```
{
    "type": "object",
    "properties": {
        "reflection": {
            "type": "string",
            "description": "A detailed analysis of the error, including the
    identified cause and an explanation of why the error occurred.",
        },
        "corrected_code": {
            "type": "string",
            "description": "The corrected code cell to be executed next that
    resolves the identified error and addresses the root cause. STRICTLY ONLY THE
    PYTHON CODE WITHOUT ANY ADDITIONAL TOKENS.",
        },
        "is_persistent_error": {
            "type": "boolean",
            "description": "Indicates whether the error is recurring based on
    previous debugging history. True if the error is persistent across executions,
    false if it’s a one-time issue.",
        },
    },
    "required": ["reflection", "corrected_code", "is_persistent_error"],
}
```

Debug Chain Integration. The prompt and json schema for the itegration of debug chain

Prompt for debug chain integration

```
You are an AI assistant helping a **ReAct-based agent** that operates using a
    **Thought-Code-Observation** loop. The agent runs code step by step in a
    Jupyter notebook, observing the output at each step.

Whenever an error occurs, a **separate debug chain** is initiated to diagnose and
    resolve the issue. This debug chain follows its own
    **Thought-Code-Observation** loop and can take up to **5 steps** to fix the
    problem.

Once the debug chain **completes** (either by fixing the issue or reaching the
    step limit), you must summarize everything that happened into **a single
    Thought-Code-Observation step**. This step will be used as the **current
    step** in the main ReAct loop, ensuring a seamless transition for the agent to
    continue execution.

---

## **You will be provided with the following:**

1. **Previous React Step**
   - The Thought-Code-Observation step where the error first occurred.

2. **Debug Chain**
   - The sequence of Thought-Code-Observation steps taken to diagnose and resolve
    the error.

---

## **Your Task:**

Based on the provided information, generate the **current step** in the ReAct loop
    using the format below:
```

```
1. **Current_Thought**:
   - **Narrative Style:** Write in **first-person perspective** to match the ReAct
    agent’s style (e.g., I observed..., I encountered...).
   - **Content Requirements:**
     - Summarize the key debugging actions taken, focusing on what occurred during
    the debug chain.
     - Clearly describe the error encountered, the debugging attempts made, and
    the final state of the code as reflected in the executed code.
     - The thought should solely serve as a reflective summary that aligns with
    the final code and observation.
   - **Tone:** Maintain a reflective, factual tone that mirrors the agent’s
    typical thought process.

2. **Current_Code**:
   - Provide the **final executed code** from the debug process.
   - If the issue was fixed, this should be the corrected version of the original
    code.
   - If the issue was **not fully resolved**, provide the **last attempted code
    version** so the agent can continue from there.

3. **Current_Observation**:
   - Summarize the **final outcome** of the debug process.
   - If the issue was fixed, indicate that the problem was resolved.
   - If the issue was **not fully resolved**, describe what remains problematic so
    the agent understands the current state before proceeding.

---

## Previous React Step:**
{previous_thought_code_observation_step}

## Debug Chain:**
{debug_chain}

---

based on the give information provide your output in the specified JSON format.
```

Json Schema for debug chain integration

```
{
    "type": "object",
    "properties": {
        "current_thought": {
            "type": "string",
            "description": (
                    "A first-person narrative summarizing the debugging process. "
                    "Describe the error encountered, the debugging attempts made,
    and the final state as reflected in the code. "
```

```
                    "This should align with the final code and observation without
    outlining future steps."
                )
            },
            "current_code": {
                "type": "string",
                "description": "The final version of the code executed after
    debugging. If the error was fixed, this should be the corrected code. If
    unresolved, this should be the last attempted version for continuity. STRICTLY
    ONLY THE PYTHON CODE WITHOUT ANY ADDITIONAL TOKENS."
            },
            "current_observation": {
                "type": "string",
                "description": "A concise summary of the final outcome of the
    debugging process. If the issue was resolved, indicate the success. If
    unresolved, describe what remains problematic so the agent can take
    appropriate next steps."
            }
        },
        "required": ["current_thought", "current_code", "current_observation"]
}
```

A.7 Examples of Intermediate Results

In this section, we present a concise overview of intermediate results, with many details shortened for brevity. For complete trajectory details, please refer to our logs.

Memory. A random example of constructed memory. This was the memory for step 5 in one of our runs.

Example of Constructed Memory

```
Here is the sequence of all Thought/Action/Action Input/Observation loops you have
    done so far for this task:

Full thought-action-action_input-observation loops, with observations summarized
    for all except the latest step):

--- Step 1 ---
Thought: I will begin by implementing our initial baseline model. The idea is to
    use a median-based predictor per coupling type. First, Ill load the training
    data and split it into training and validation sets by molecule (using an
    80/20 split) to mimic the competitions molecule-based train/test split. On the
    training split I will compute the median scalar_coupling_constant for each
    coupling type. Then, for our validation set I will predict the median (based
    on the coupling type) and compute the baseline validation metric using the
    competitions metricspecifically, for each coupling type, the log of the mean
    absolute error (with a floor of 1e-9) is computed, and the final score is the
    average of these per-type scores.

After checking our baseline validation score, I will use the entire training data
    to compute the medians (per coupling type) and then predict on the provided
    test set. Finally, Ill create a submission file named exactly
    "submission_{baseline_metric}.csv" (with the metric rounded to 4 decimals)
    that contains the test predictions.

Action: python_code_executor
Action Input:
‘‘‘python
import pandas as pd
import numpy as np

# Load data
train_df = pd.read_csv("input/train.csv")
test_df = pd.read_csv("input/test.csv")

# Create a validation split by grouping on unique molecules (80% training, 20%
    validation)
...
[Removed for brevity]
...
‘‘‘
Observation:
The script computed a baseline validation score of approximately 1.1859 by
    calculating the mean log MAE from the validation split and then used medians
    from the full training data to create predictions for the test set. A
    submission file named submission_1.1859.csv was successfully generated, and
    the entire process completed without any errors in about 4 seconds.


--- Step 2 ---
Thought: I will now improve the baseline by adding a physics-inspired feature  the
    distance between the two atoms  as computed from the molecular structure (from
    the structures.csv file). Ill merge structures with train and test data to
    compute the Euclidean distance between the two atoms. Then, for each scalar
    coupling type, I will train a simple linear regression model (using distance
    as the sole predictor) on a training split (80% of molecules) and evaluate it
    on a validation split (20% of molecules) using the competition metric.
    Finally, Ill retrain on the full training data and generate predictions for
    the test set, saving the submission file as "submission_{lr_metric}.csv"
    (using the new metric rounded to 4 decimals).

Action: python_code_executor
Action Input:
‘‘‘python
...
[Removed for brevity]
...
‘‘‘
Observation:
The script successfully computed the validation score using Linear Regression
    (score: 1.0169) but then encountered a ValueError when predicting on the test
    set due to NaN values in the ’distance’ column. This error prevented the
    creation of the submission file, and the output message provides suggestions
    to handle missing data.
```

```
--- Step 3 ---
Thought:
I encountered the error because the test dataset contained NaN values in the
    ’distance’ feature, stemming from missing coordinate values after the left
    merge with the structures dataframe. To resolve this, I imputed the missing
    ’distance’ values in the test set using the median value computed from the
    training data. I then reran the model training and prediction pipeline for
    each coupling type, which allowed the LinearRegression model to run without
    error and produce a valid submission file.
Action:python_code_executor
Action_input:
‘‘‘python
...
[Removed for brevity]
...
‘‘‘

Observation:
The error was resolved successfully. The final code imputes missing ’distance’
    values in the test set using the median value from the training data, which
    allowed the LinearRegression model to run without errors. The validation score
    was computed (approximately 1.0169) and the submission file
    ’submission_1.0169.csv’ was created.
Execution summary of the code:
The execution successfully preprocessed the data, computed the Euclidean distances
    and validation score (approximately 1.0169), and trained linear regression
    models per coupling type. A submission file named ’submission_1.0169.csv’ was
    created as expected with no errors.


--- Step 4 ---
Thought: In this step, I will try a new approach: instead of using linear
    regression based solely on the distance feature, I will now use a more
    flexible model. I will train a HistGradientBoostingRegressor (a fast
    tree-based method from scikit-learn) for each coupling type using the distance
    feature. As before, I will merge the structures dataframe with the train and
    test data to compute the interatomic distance, impute any missing values in
    test using the median distance from train, and then split the train data by
    molecules (80/20 split) to obtain a validation score. Finally, I will retrain
    the HistGradientBoostingRegressor models on the full training dataset and
    generate a test-set prediction file named submission_{hgb_metric}.csv (with
    the metric rounded to 4 decimals).

Action: python_code_executor
Action Input:
‘‘‘python
...
[Removed for brevity]
...
‘‘‘
```

```
Observation:
[SYSTEM NOTIFICATION] Running the above tool with the following response:
Terminal output:
HistGradientBoostingRegressor validation score: 0.7639257173693997
Submission file created: submission_0.7639.csv

Execution time: 21 seconds (time limit is 9 hours).

--- Step 5 ---
```

Main Agent. An example of Main Agent response for one of our runs at a random step.

Example of Main Agent response

```
----------------------------
Competition : rsna-miccai-brain-tumor-radiogenomic-classification
Modality    : Image
Result      : GOLD medal
Current Step: 18
----------------------------
User Prompt:
----------------------------
Answer the following questions as best you can.
You have access to the following tools:
1. python_code_executor[input]: this tool can help to run python script with
    python code as input

You are participating in *MLE-bench*, an offline version of Kaggle competitions
    adapted to test the coding ability of agents like yourself.

You will be given a machine learning task. You must solve the task by training a
    model and running the model on the test set to produce a submission file. You
    may not hand-label the data to produce a submission file.

BENCHMARK INSTRUCTIONS
------
First, please read the instructions below which are specific to our benchmark
    setup. Note, these instructions can be found again in ‘instructions.txt‘.

- **Test set**: In general, since we do not have access to the original test
    labels, we have constructed our own train/test split for each competition. As
    such, the dataset you see in ‘/input‘ may differ slightly from what is
    described in the competition instructions. Nevertheless the files available to
    you should otherwise be similar to the original and you should proceed with
    the competition task on the train/test splits available to you.
- **Cheating**: You are not allowed to view others’ solutions to this competition.
    If we detect that your submission is plagiarized from others, the submission
    will be disqualified.

Your submission will be graded based on the metric in the competition-specific
    instructions. Your goal is to achieve the best score possible on the
    competition task.
```

```
Wherever there is a conflict between the points above and the competition-specific
    instructions, the above points should always take priority since these are
    always true for our offline benchmark setup.

ADDITIONAL NOTES
------

- **Compute**: You have access to 1 NVIDIA A100 80GB PCIe with the appropriate
    drivers installed.
- **Use as much of the available time as possible to refine and optimize your
    submission**: Your goal is to achieve the best possible score, not just to
    submit your first working solution. Unless you are certain that no further
    improvements can be made. Manually spot-checking predictions, building custom
    validation sets and grading functions, and comparing different algorithms. You
    may keep working until the time limit expires -- as long as your best
    submission is stored at the designated location at the end of your run, it
    will be picked up for grading.

COMPETITION INSTRUCTIONS
------

# RSNA-MICCAI Brain Tumor Radiogenomic Classification

...
[Full Competition Page including Description, Evaluation, Submission File, Code
    Requirements, Dataset Description. Removed for brevity]
...

All the required data has been uploaded by the user in the current directory.
    IMPORTANT NOTE: If any zip files were uploaded, they have already been
    unzipped, and their contents are ready to use.
Data directory preview:
‘‘‘
input/
    sample_submission.csv (60 lines)
    train_labels.csv (527 lines)
    test/
        00002/
            FLAIR/
                Image-387.dcm (525.4 kB)
                Image-388.dcm (525.4 kB)
                Image-389.dcm (525.4 kB)
                Image-390.dcm (525.3 kB)
                ... and 125 other files
            T1w/
... (truncated)
‘‘‘
No additional unzipping or extraction is required. Please proceed with the data
    as-is.
```

```
**Your task is to create an initial baseline validation metric and generate a
    corresponding submission_{validation_metric.csv} file. Then, continuously
    improve the validation metric and produce updated
    submission_{validation_metric}.csv files for each improvement.**

Always respond in this format exactly:
Thought: What you are currently doing, what actions to perform and why
Action: the action to take, should be exactly one name of the tool from this list
    - [python_code_executor]
Action Input:
‘‘‘python
[the input to the action]
‘‘‘
Observation: the result of the action
... (this Thought/Action/Action Input/Observation should continue repeating
    indefinitely until instructed to stop)

TASK INSTRUCTIONS:
...
[Same task instructions as provided in the appendix for prompts. Removed for
    brevity]
...

-------
COMPUTATIONAL AND RESOURCE GUIDELINES:
1. TIME MANAGEMENT:
   - TOTAL AVAILABLE TIME: 24h 0m 0.00s
   - TIME SPENT: 22h 51m 21.23s
   - TIME REMAINING: 1h 8m 38.77s

2. ADAPTIVE STRATEGY:
   - If REMAINING TIME < 25% of TOTAL:
     * SKIP non-essential steps
     * PRIORITIZE core functionality
   - Dynamically adjust complexity based on remaining time
   - Focus on critical tasks; defer optional enhancements if needed

NOTE: Both CPU and GPU is present and most of the necessary libraries are
    installed. Efficient use of resources will help ensure task completion within
    the allocated limits.
-------

MAKE SURE YOU FOLLOW THE INSTRUCTIONS WHILE EXECUTION.

Steps 8 to 17 (Showing only action inputs. Previous steps truncated due to memory
    limits):

# %%
#Action:python_code_executor

#Action Input:
import os
import numpy as np
import pandas as pd
import pydicom
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import StratifiedKFold, cross_val_predict
from sklearn.metrics import roc_auc_score

def extract_median_slice_features(subject_path, modality):
    """
    Extracts features from the median DICOM slice of the given modality:
    """
...
...
[Rest of the memory removed for brevity]
...
--- Step 18 ---
Thought:
(Begin!)

----------------------------
Main Agent Response:
----------------------------
THOUGHT: I want to improve the CV ROC-AUC by increasing the PCA dimensionality for
    the CNN features from 200 to 300. This should better preserve the variance in
    the CNN features before fusing with the handcrafted ones. I’ll re-run the
    pipeline with PCA(n_components=300) and then perform hyperparameter tuning
    using GridSearchCV with XGBoost, followed by generating a new submission file.

ACTION: python_code_executor

ACTION INPUT:‘‘‘python
import os
import numpy as np
import pandas as pd
import pydicom
from PIL import Image
import torch
import torchvision.transforms as T
import torchvision.models as models
from xgboost import XGBClassifier
from sklearn.decomposition import PCA
from sklearn.model_selection import StratifiedKFold, GridSearchCV,
    cross_val_predict
from sklearn.metrics import roc_auc_score
...
[Removed for brevity]
...
submission_file_name = f"submission_{cv_auc:.3f}.csv"
submission_df.to_csv(submission_file_name, index=False)
print("Submission file created:", submission_file_name)
‘‘‘
```

Summary Agent. An example of Summary Agent response for one of our runs at a random step.

Example of Summary Agent response

```
-------------------
OBSERVATION
-------------------
[SYSTEM NOTIFICATION] Running the above tool with the following response:
Terminal output:

tokenizer_config.json:   0%|          | 0.00/48.0 [00:00<?, ?B/s]
tokenizer_config.json: 100%|##########| 48.0/48.0 [00:00<00:00, 392kB/s]
...
[Rest of the 389 lines truncated for brevity here]
...
 93%|#########3| 115/123 [00:04<00:00, 28.07it/s]
 96%|#########5| 118/123 [00:04<00:00, 28.08it/s]
 98%|#########8| 121/123 [00:04<00:00, 28.13it/s]
100%|##########| 123/123 [00:04<00:00, 28.43it/s]
Submission file saved as submission_0.3799.csv

Execution time: 8 minutes (time limit is 9 hours).


-------------------
OBSERVATION SUMMARY
-------------------
{
    ’is_bug’: False,
    ’has_csv_submission’: True,
    ’submission_file_name_list’: [’submission_0.3799.csv’],
    ’summary’: "The model, implemented using a BERT-based classifier and a custom
    dataset, was successfully trained for 3 epochs, with the final validation log
    loss reported as approximately 0.3799. Training and evaluation metrics,
    including a training loss of about 0.2832 and high evaluation throughput (~440
    samples per second), were output, and a submission CSV file named
    ’submission_0.3799.csv’ was saved. A minor warning about newly initialized
    classifier weights was displayed, indicating that further task-specific
    fine-tuning is recommended."
}
```

Debug Chain. An Example of debug chain in one of our runs.

Example of debug chain

```
----------------------------
OBSERVATION WITH ERROR
----------------------------
[SYSTEM NOTIFICATION] Running the above tool with the following response:
Terminal output:

MC Dropout TTA_v1 Validation Iteration 1/5:   0%|          | 0/314 [00:00<?, ?it/s]
```

~~~
MC Dropout TTA_v1 Validation Iteration 1/5:   0%|          | 1/314 [00:03<20:43,
    3.97s/it]
...
[Rest of the observation was truncated for brevity]
...
MC Dropout TTA_v2 Validation Iteration 5/5: 100%|#########9| 313/314 [03:22<00:00,
     1.79it/s]
MC Dropout TTA_v2 Validation Iteration 5/5: 100%|##########| 314/314 [03:22<00:00,
     1.55it/s]
Error in line 138: ensemble_val_auc_mcdrop = roc_auc_score(all_val_labels,
    ensemble_val_preds)
ValueError: Found input variables with inconsistent numbers of samples: [32, 10033]
Execution time: 41 minutes (time limit is 9 hours).

----------------------------
OBSERVATION SUMMARY
----------------------------
The execution output shows extensive progress of MC Dropout TTA on both validation
    and test datasets, but it ultimately fails with a ValueError due to
    inconsistent sample sizes (32 vs 10033) when computing the ROC AUC score. As a
    result, no CSV submission file was created, and the error halted further
    execution after approximately 41 minutes.

----------------------------
---DEBUG MODE (separate chain initiated)---
----------------------------
DEBUG STEP-1
----------------------------
REFLECTION 1:
The error occurred because the ground truth labels (all_val_labels) were being
    incorrectly accumulated from only a single batch of the validation data rather
    than from the entire validation set. In the original code, the labels were
    captured inside the TTA loop using 'if all_val_labels is None: all_val_labels
    = np.array(labels.numpy())', which resulted in all_val_labels having the
    length of a single batch (e.g., 32) instead of the full set (e.g., 10033
    samples). This mismatch causes the roc_auc_score function to raise a
    ValueError for inconsistent number of input samples. This error seems to be
    isolated to this MC Dropout ensemble TTA block (it has not been a recurring
    error in earlier cells).

CORRECTED CODE 1:
```python
...
[Removed for brevity]
...
```

EXECUTION OUTPUT 1:
Terminal output:

MC Dropout TTA_v1 Validation Iteration 1/5:   0%|          | 0/314 [00:00<?, ?it/s]
~~~

```
MC Dropout TTA_v1 Validation Iteration 1/5:   0%|          | 1/314 [00:03<18:10,
    3.48s/it]
MC Dropout TTA_v1 Validation Iteration 1/5:   2%|1         | 5/314 [00:05<05:24,
    1.05s/it]
...
[Rest of the observation was truncated for brevity]
...
MC Dropout TTA_v2 Test Iteration 5/5: 100%|##########| 1121/1121 [15:07<00:00,
    1.23it/s]
Submission file with MC Dropout Ensemble TTA saved as:
    submission_0.9352_tta_mcdrop.csv

Execution time: 3 hours (time limit is 9 hours).

----------------------------

[HERE ERROR WAS RESOLVED ---> BREAKING OUT OF DEBUG CHAIN]

----------------------------
DEBUG CHAIN INTEGRATED STEP
----------------------------
THOUGHT:
I identified that the ValueError was due to an inconsistent accumulation of
    validation labels; originally, I was only capturing labels from a single
    batch, leading to a mismatch with the predictions generated from the entire
    dataset. To fix this, I modified the code to accumulate all labels from the
    full validation set prior to running the TTA loops. I then verified that both
    TTA iterations (using transform v1 and v2) work correctly and the final
    ensemble AUC calculation now receives matching inputs. The submission file is
    generated without errors.

ACTION: python_code_executor

ACTION INPUT:
‘‘‘python
...
[Removed for brevity]
...
‘‘‘

OBSERVATION & OBSERVATION SUMMARY:
The script executed a Monte Carlo Dropout with Test Time Augmentation (TTA)
    approach for both the validation and test datasets. It iteratively computed
    predictions using two different augmentation transforms, ensemble averaged
    them, and achieved a validation AUC of approximately 0.9352. Finally, it
    successfully saved a submission file named ’submission_0.9352_tta_mcdrop.csv’.

----------------------------
---DEBUG MODE (finished)---
----------------------------
NEXT STEP (by Main Agent)
...
```

Submission Checklist

1. For all authors…

   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes] [We support our claims with detailed results and analysis along with the Code and logfiles.]

   (b) Did you describe the limitations of your work? [Yes] [We talk about the limitations in Section [3.6]]

   (c) Did you discuss any potential negative societal impacts of your work? [No]

   (d) Did you read the ethics review guidelines and ensure that your paper conforms to them? (see `https://2022.automl.cc/ethics-accessibility/`) [Yes]

2. If you ran experiments…

   (a) Did you use the same evaluation protocol for all methods being compared (e.g., same benchmarks, data (sub)sets, available resources, etc.)? [Yes]

   (b) Did you specify all the necessary details of your evaluation (e.g., data splits, pre-processing, search spaces, hyperparameter tuning details and results, etc.)? [Yes]

   (c) Did you repeat your experiments (e.g., across multiple random seeds or splits) to account for the impact of randomness in your methods or data? [Yes] [We have reported evaluations on different splits of the data to consider for the impact of randomness.]

   (d) Did you report the uncertainty of your results (e.g., the standard error across random seeds or splits)? [No]

   (e) Did you report the statistical significance of your results? [Yes]

   (f) Did you use enough repetitions, datasets, and/or benchmarks to support your claims? [Yes]

   (g) Did you compare performance over time and describe how you selected the maximum runtime? [Yes]

   (h) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes]

   (i) Did you run ablation studies to assess the impact of different components of your approach? [Yes]

3. With respect to the code used to obtain your results…

   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results, including all dependencies (e.g., `requirements.txt` with explicit versions), random seeds, an instructive `README` with installation instructions, and execution commands (either in the supplemental material or as a URL)? [Yes]

   (b) Did you include a minimal example to replicate results on a small subset of the experiments or on toy data? [Yes] [We show that users can use any of the run groups which are small subset of the MLE-Pi dataset in the README.md file in the code.]

   (c) Did you ensure sufficient code quality and documentation so that someone else can execute and understand your code? [Yes]

   (d) Did you include the raw results of running your experiments with the given code, data, and instructions? [Yes] [We have included all the runtime logfiles and result jsons for our exeperiments]

   (e) Did you include the code, additional data, and instructions needed to generate the figures and tables in your paper based on the raw results? [Yes]

4. If you used existing assets (e.g., code, data, models)...
   (a) Did you cite the creators of used assets? [Yes]
   (b) Did you discuss whether and how consent was obtained from people whose data you're using/curating if the license requires it? [Yes]
   (c) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]

5. If you created/released new assets (e.g., code, data, models)...
   (a) Did you mention the license of the new assets (e.g., as part of your code submission)? [Yes]
   (b) Did you include the new assets either in the supplemental material or as a URL (to, e.g., GitHub or Hugging Face)? [Yes]

6. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to institutional review board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

7. If you included theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [Yes]
   (b) Did you include complete proofs of all theoretical results? [Yes]