Scaling Inference Time Compute for Machine Learning Engineering Agents

Asim Osman (asim@aims.ac.za)
African Institute for Mathematical Sciences (AIMS)
University of Cape Town, South Africa

Supervised by: Dr. Arnu Pretorius, Arnol Fokam
University of Stellenbosch, South Africa & InstaDeep, South Africa

12 June 2025

Submitted in partial fulfillment of an AI for science masters degree at AIMS South Africa

Abstract

Problem and Motivation: The high cost and limited accessibility of proprietary large language models (LLMs) for machine learning engineering tasks has created a significant barrier for researchers and practitioners. While recent inference-time scaling techniques have shown promise in enhancing model capabilities without requiring larger parameters, their application to coding and ML engineering domains—particularly with open-source models—remains largely unexplored. Approach: This work presents the first systematic implementation of inference-time scaling (ITS) strategies within an open-source agentic framework for ML engineering tasks. We augment the AIDE (AI-Driven Exploration) agent scaffold with multiple ITS techniques including self-consistency, iterative self-debugging, self-reflection, and modular task decomposition, specifically targeting DeepSeek-R1 distilled models (7B, 14B, and 32B parameters). We evaluate these enhanced agents on a curated subset of MLE-Bench competitions spanning diverse ML domains. Key Findings: Our experiments reveal 10% improvement in valid submission rate, 10% increase in medal-winning performance. Notably, we observe that ITS effectiveness correlates strongly with model size—the 32B DeepSeek model shows substantial gains from our strategies, achieving performance comparable to GPT-4 Turbo, while smaller 7B models demonstrate limited improvement. Self-consistency and prompt chaining prove most effective, while self-reflection shows minimal gains, suggesting limitations in these models' self-correction capabilities. [PLACEHOLDER: Specific numerical comparisons with baselines] Impact and Novelty: This work delivers the first open-source, accessible framework for ML engineering automation that demonstrates competitive performance with proprietary alternatives. We provide four distinct AIDE variants, each optimized for different ITS strategies, establishing a foundation for future research in open-source coding agents. Our findings offer crucial insights into the scaling behavior of inference-time techniques in coding domains, demonstrating that strategic computational allocation during inference can bridge the capability gap between open-source and proprietary models.

Declaration

I, the undersigned, hereby declare that the work contained in this research project is my original work, and that any work done by others or by myself previously has been acknowledged and referenced accordingly.

Asim Osman, 12 June 2025

Contents

1. Introduction

Over the last few years, advancements in large language models have primarily relied on scaling up the number of parameters alongside increasing dataset size and compute budgets. This strategy, often referred to as train-time compute scaling, has been remarkably effective, significantly and consistently improving model performance on tasks such as translation, general question answering, and sophisticated reasoning in mathematics and coding. However, the exponential growth in model size and the associated costs have become prohibitively expensive, pushing training budgets to the billion-dollar scale and raising concerns about resource sustainability, particularly as high-quality training data becomes scarcer.

Consequently, significant interest has shifted toward inference-time (or test-time) scaling techniques. Rather than focusing on costly retraining and larger models, inference-time scaling enhances existing models by dedicating additional computational resources during prediction. These methods enable models to "think longer" on challenging problems by dynamically generating multiple reasoning paths, employing voting mechanisms, or iteratively refining their outputs. Recent studies, such as DeepMind's compute-optimal scaling (Doe, 2023), demonstrate how smaller models can achieve impressive performance through adaptive inference strategies, sometimes even outperforming substantially larger models.

In parallel with these developments, there has been a notable emergence of "reasoning" or "thinking" models, designed explicitly to enhance models' capabilities to reason through complex problems. A prime example is DeepSeek-R1, which explicitly trains models using reinforcement learning to produce long, structured chains of thought before providing an answer, enabling LLMs to systematically tackle tasks that were previously challenging even for large-scale models. DeepSeek released a series of small distilled models trained on reasoning datasets generated from the original R1 model, empowering them with remarkable reasoning capabilities and improving their performance relative to larger models.

Motivated by these developments, this work explores the promise of inference-time scaling techniques applied to small-scale language models-particularly those distilled from DeepSeek-that are specifically designed for coding and machine learning engineering tasks, areas still challenging even for frontier models. Smaller models have substantial advantages in accessibility, rapid deployment, and lower operational costs, but traditionally lack the advanced reasoning capabilities exhibited by larger models. Through systematic implementation and evaluation of strategies such as chain-of-thought prompting, iterative self-refinement, self-consistency, and other methods. Ultimately, this thesis evaluates the extent to which inference-time scaling can address complex tasks such as machine learning engineering. We demonstrate that, with carefully designed inference-time enhancements, small-scale models can significantly improve performance in complex coding and engineering contexts, presenting a viable and cost-effective alternative to training massive foundational models.

2. Background and Related Work

2.1 Background

2.1.1 Large Language Models (LLMs)

Large Language Models (LLMs) are neural networks trained on extensive text datasets, capable of generating human-like text. Based primarily on the transformer architecture, these models utilize a straightforward predictive objective, leveraging enormous volumes of data to tackle complex problems and diverse natural language tasks [CITATION], including translation [CITATION], semantic analysis [CITATION], and information retrieval.

Recent advancements in LLMs, exemplified by models such as the GPT series [CITATION*3] and the Llama family [CITATION], have been driven by significant scale, reaching billions of parameters. This substantial scale has unlocked emergent abilities—advanced skills like reasoning and in-context learning—that smaller models typically do not exhibit. These emergent capabilities enable LLMs to solve arithmetic problems, answer intricate questions, and summarize complex passages, often without explicit task-specific training.

2.1.2 Scaling Laws and Train-Time Compute

Scaling lows shows how the performance of Large Language models improves in a predeictable way with the increase in the parameter size, training data and computationa; resources, openai ahs shown that LLMs performance as measured by metrics such as accuracy or perplexity follows a power low relationship with these scaling factors. Performance enhancements are steady yet exhibit diminishing returns as scale increases, motivating the development of increasingly larger models like PaLM and Chinchilla.

While scaling laws provided performance guarantees and drove a competitive race for building larger models, leading to current models in the hundreds of billions of parameters, practical limitations have surfaced. High-quality data is becoming scarce relative to growing model demands, and the significant costs and resource consumption associated with these models pose substantial challenges. Moreover, LLMs auto-regressive and token level prediction style is itself a limitation when it comes to complex reasoning tasks such as advanced mathematics or complex question answering. While recent years witnessed substantial developments in the models abilities, especially using post training and fine tuning, this requires massive computational resources and large scale datasets, encouraging researchers to to find cheaper approaches.

2.1.3 Inference-Time Scaling (ITS)

To address limitations associated with traditional scaling strategies, a new paradigm has emerged: inference-time scaling, also known as test-time scaling. Instead of allocating more resources to pre-training, this approach dynamically adjusts computational resources during inference, enabling models to “think longer” and thus handle more complex problems. Inference-time scaling involves techniques such as generating longer reasoning chains, sampling multiple possible answers, and iteratively refining outputs. This strategy has shown significant promise, allowing smaller models to surpass larger counterparts.

Techniques in inference-time scaling include:

- **Chain-of-Thought Prompting (CoT):** Encouraging models to explicitly generate intermediate reasoning steps.
- **Iterative Self-Refinement:** Models iteratively identify and correct errors in their outputs.
- **Self-Consistency:** Generating multiple independent outputs and selecting the most consistent result via voting.
- **Verifier-Guided Search Techniques:** Utilizing reward-based verifiers or structured search methods to identify optimal answers from multiple generated solutions.

2.1.4 Emergence of Reasoning Models

Recent developments have highlighted the emergence of models focused on reasoninging specifically designed to enhanthe capabilities in multistep logical and mathematical reasoning. These reasoning models utilize inference-time strategies like structured prompting, external tool usage, and systematic search without altering model weights. Techniques such as ReAct, PAL, Toolformer, and Tree-of-Thoughts have demonstrated significant improvements in reasoning abilities.

DeepSeek-R1 serves as a notable landmark, explicitly trained via reinforcement learning on structured reasoning datasets to generate coherent reasoning chains, significantly improving performance on complex reasoning tasks. DeepSeek's release of smaller distilled models, trained on datasets generated by R1, demonstrates the potential of distillation combined with structured inference strategies, achieving performance comparable or superior to larger models.

* the effect of rl training on producing long chains of thoughts * the distilled models that were trained on the reasoning datata from R1 model

2.1.5 Agentic Systems and Scaffolding

"Scaffolding" reffers to code built up around an LLM in order to augment its capabilities. This does not typically include code which alters the LLM's internals, such as fine-tuning or activation steering. People use scaffolding because it can allow LLMs to use tools, reduce error rates, search for info, all of which make LLMs more capable. Common types of scaffolds include: Prompt templates: Perhaps the simplest scaffolds, "prompt templates" are just prompts with some blank sections to be filled in at runtime using some local data. E.g., a template like "You are a good assistant. The date is INSERT DATE HERE. Tell me how many days are left in the month." can be filled in with the current date before being used to prompt an LLM. Retrieval Augmented Generation (RAG): For tasks which have some associated data, RAG is a way of automatically adding task-relevant info to prompts. RAG does this by looking for snippets of text/data in the database which are relevant to the user's prompt, and then adding those snippets to the prompt. Search engines: Similar to RAG, giving an LLM access to a search engine lets it find info relevant to its prompt. Unlike RAG, the LLM decides what to search for and when. Agent scaffolds: Scaffolds which try to make LLMs into goal-directed agents, i.e., giving an LLM some goal/task to perform, and ways to observe and act on the world. Usually, the LLM is put into a loop with a history of its observations and actions, until its task is done. Some agent frameworks give LLMs access to much more powerful high-level actions, which saves the AI from repeating many simple primitive actions and, therefore, speeds up planning.

Agent scaffolds are characterized by their capacity for perception, planning, and action within an environment, often governed by an iterative process that refines, reflects, and improves the solution over multiple cycles.

While agents scaffolds can be used in different domains and wide array of tasks, there has been increasing interest dedicated to developing agents capable of automating the machine learning research and engineering process. A notable example is the [AI scientist] system, which presents an end-to-end pipeline designed to automate the research workflow. Currently, data science, machine learning engineering, and more generally, coding agents and assistants have shown consistent improvements. as we now see agents like Gihub Capilot, Cognition | Introducing Devin, Cursor, or Claude code exhibit phenomenon coding abilities, but there is no significant progress in their open-source coding agents.

The nature of coding problems presents a unique challenge for LLMs compared to other reasoning tasks, such as mathematics. In mathematical reasoning, there is often a single, unambiguously verifiable answer that can be derived through logical deduction. Conversely, generating a coding solution involves not only producing syntactically correct and executable code, but also navigating a vast solution space, where multiple functionally equivalent but structurally diverse implementations can exist. Crucially, the correctness and optimality of a coding solution often necessitate empirical validation through actual code execution and looking at traceback, rather than purely symbolic or logical verification. This inherent complexity, which demands iterative refinement and empirical testing, underscores the critical need for sophisticated agentic systems that can autonomously generate, execute, and iteratively refine solutions, even for highly capable foundational models.

2.1.6 Relevance and Challenges in Coding and ML Engineering Tasks

Machine learning engineering and coding tasks represent particularly challenging yet highly relevant applications for inference-time scaling due to their inherent complexity and precision requirements. Benchmarks like HumanEval, MBPP, and MLE-Bench exemplify these tasks, providing challenging evaluation contexts.

However, significant gaps persist in current research, such as limited open-source implementations and insufficient evaluation on realistic, complex coding tasks. This project explicitly addresses these gaps by systematically applying inference-time scaling techniques to small-scale distilled reasoning models, aiming to substantially enhance their performance and practical applicability in coding and ML engineering domains.

CITATION NEEDED

2.2 Related Work

3. Methodology and Experimental Setup

In this chapter, we describe the experimental setup and the design choices made in adapting and building our agent scaffold upon existing frameworks, and the specific procedures for evaluating the proposed inference-time scaling (ITS) strategies. The primary objective is to establish a replicable evaluation framework that can quantitatively assess the impact of these ITS techniques when applied to open-source Large Language Models (LLMs) in complex machine learning engineering tasks.

To achieve this, we first describe the main components of our experimental environment: the AI-Driven Exploration (AIDE) agent scaffold, which forms the basis for our agent development, and the MLE-Bench benchmark, which provides the tasks for evaluation. We then elaborate on the significant engineering efforts undertaken to adapt these tools for effective use with locally hosted open-source LLMs, including the development of a custom vLLM-based backend for AIDE and a parallelized execution environment for MLE-Bench experiments. Finally, we outline the chosen baseline models, the specific subset of MLE-Bench competitions used, the evaluation metrics, and the overall experimental pipeline designed to ensure fair comparisons.

3.1 AIDE: AI-Driven Exploration in the Code Space

AIDE is an open source agent scaffold, developed by WecoAi (Schmidt et al., 2025). It is specifically designed to automate the trial and error process in developing machine learning models, working iteratively to find a solution in a tree search manner, exploring the ‘Code space‘.

The core principle behind AIDE is to address the significant amount of time that engineers and scientists spend on iterative experimentation, rather than to focus on conceptualizing and innovating. To achieve that, AIDE frames the entire machine learning engineering process as a code optimization problem, modeling the trial and error as a tree search within a space of potential code solutions, each node is a potential solution problem (i.e. a code script), and each edge is an attempt at improving/debugging that solution. figure 3.1 shows a sample solution tree for AIDE.

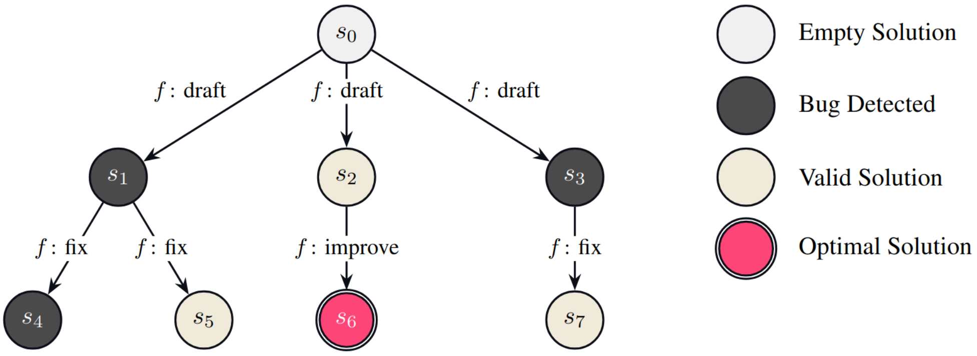

Figure 3.1: A sample solution tree for AIDE, illustrating the drafting, debugging, and improvement steps. (Source: Schmidt et al., 2025, p. 4)

At its heart. and as many other agent scaffolds, AIDE is powered by a Large Language Model, typically a strong model like gpt4 or Claude. These LLMs are responsible for proposing new code, debugging existing scripts, or suggesting and improvement or refinement to promising solutions. AIDE then reuses and refines these solutions, effectively trading computational resources for improved performance. Essentially, AIDE is performing some sort of inference time scaling, as it create an iterative loop of drafting, debugging and improving code solutions but the scope and the level of this scaling is not as intensive. The implementation of AIDE is publicly available, which is why we chose it for this thesis, allowing integration of custom inference time scaling methods.

3.1.1 Core Components and Methodology of AIDE

Below is a breakdown of how AIDE operates as outlined in their original paper(Schmidt et al).

The Solution Tree (T): This is all discovered/proposed solutions (Python code scripts) and the improvement attempts (edges linking them) stored in a tree structure. The root solution (s_0) is typically an empty script, a node with value 'None'. The tree is used to store the history of the search process. In code this is typically saves as a Journal file (json) which documents the entire search process.

Evaluator (h): This is a stateless function that takes a solution script as input and returns a scalar score (e.g., validation accuracy, loss). This score guides the search process. this evaluator is composed of a code interpreter that executes the code, then another LLM then takes the execution result (the traceback) and produces a structured output via function calling. this particular role is considered stateless since it always produce the same score, and it does not include past attempts in its evaluation, and this is important in the overall tree search for the best solution as every node or possible solution is evaluated primary on this.

Search Policy (π): AIDE uses a simple, hard-coded search policy that determines the next step or next action. Based on the current state of the solution tree, it decides to:

1. Draft a new initial solution (if a desired number of diverse starting points hasn't been reached). At the beginning of the search, AIDE will draft a number of initial solution that will represent an initial exploration of the solution space, this is done by prompting the LLM to generate a plan to solve the problem and then generate a single-file self contained Python program implementing that plan. These are then separated using regular expressions. The llm is also given a data preview of the dataset, and a set of instructions describing the desired output format and the metric to be used for evaluation.

In the drafting phase, AIDE asks the llm to generate a simple solution, without any optimization or hyperparameter tuning, this is done to ensure that the solution is valid and working, and to get a baseline score to compare against. nd allow fro easy debugging and improvement.

2. Debug a buggy node (if it's within a certain debug depth). If the draft solution is not working, AIDE will debug it by prompting the LLM to inspect the and execution traces which is reformatted into a structured output, and then generate a plan to fix the script and the updated solution is then evaluated. The debug is controlled by a probability parameter, and the default is set to 1, meaning that 100% of the time, AIDE will debug if there is something to debug, although this can be changed to a lower value to encourage more exploration and avoid getting stuck in a local optimum.

3. Improve an existing, non-buggy solution (typically targeting the current best-performing one). In the improvement phase, AIDE will prompt the LLM to propose a single 'atomic' change, such as switching an optimizer or adding a regularization technique, so its impact on performance can be directly measured. AIDE will keep asking for improvements until the time limit is reached, or the maximum number of steps is reached.

Coding Operator or coder model (F): This is the LLM itself as it proposes new scripts. It has three main entry points described in the search policy, Drafting, Debugging and Improving, each with specialized prompts. these coding agents are facilitated using modular backend script for each model provider like

Openai, Anthropic, Google, or Meta. and the switch between them is done using a dispatcher logic that routes the request to the appropriate model provider based on the model id.

Data Preview in Coding Prompts: AIDE also includes a small, static 'data preview' in its coding prompts, providing the LLM with basic knowledge of the dataset's size or feature layout without needing extensive exploratory data analysis (EDA) at each step. As there is a code interpreter part of this iterative process, this Data preview tells the model exactly where and how the data is organized, and where to save its output or 'submission.csv'

Finally, to manage the LLM's context window and avoid 'prompt explosion' from an the growing history being fed to the model during the debugging and improvement phases, AIDE uses a summarization operator. Instead of appending all historical logs, this operator selectively extracts relevant information from the solution tree, such as Performance metrics (accuracy, AUC-ROC, etc.), Hyperparameter settings from previous attempts, Relevant hints for debugging (e.g., misaligned array shapes from tracebacks), This concise summary allows each code revision to be somewhat stateless yet guided by prior information, maintaining efficiency.

The choice of AIDE as the foundational scaffold for this research is deliberate. It is open-source , it allows for the modification and integration of the inference-time scaling methods (self-consistency and self-reflection, etc) that are central to our work. Furthermore, AIDE's design, which explicitly models ML engineering as an iterative code optimization and tree search process powered by an LLM, provides a structured environment to study the effects of these techniques. It is established and proven to work effectively in solving ML tasks, as demonstrated in its original paper and subsequent evaluations on benchmarks like MLE-Bench, offers a great platform for experimentation.

AIDE has been tested and evaluated only using lagre-scale propriety LLMs, and our aim is to investigate how inference-time strategies can enhance AIDE's problem-solving capabilities, particularly its ability to make an open source, small-scale LLM generate more robust and higher-performing solutions on the challenging task like Machine learning Engineering.

A typical AIDE run usually starts by generating a draft plan and a code script implementing that plan, then the code is executed and the result is evaluated and stored as a node in the solution tree, if the code is not working, the node is marked 'buggy', then the agent proceeds to the next step, queying the search policy, which first looks at the number of initial drafts, if its less than the specified number in the configuration, the decesiopn will be to draft a new solution, otherwise the search policy will look first at the good nodes, if there is any, it acts greedly and tries to improve it, if there is no good node, it decides to debug a buggy code depending on a probability parameter. The code is debugged and the process is repeated until time limit is reached, or the maximum number of steps is reached.

3.2 MLE-Bench: A Benchmark for Machine Learning Engineering

3.2.1 MLE-bench Design

The core design of MLE-Bench is mostly around its task selection. The 75 selected competitions present an extremely difficult challenge for any coding agent. As per the authors, this benchmark focuses on two design choices: i. selecting tasks that are challenging and representative of contemporary ML engineering work, and ii. being able to compare evaluation results to human-level performance.

Essentially, MLE-Bench provides an offline Kaggle competition environment. For context, Kaggle is a platform that hosts data science and ML competitions where participants build predictive models to solve real-world challenges, competing for the best score on predefined metrics and earning rankings

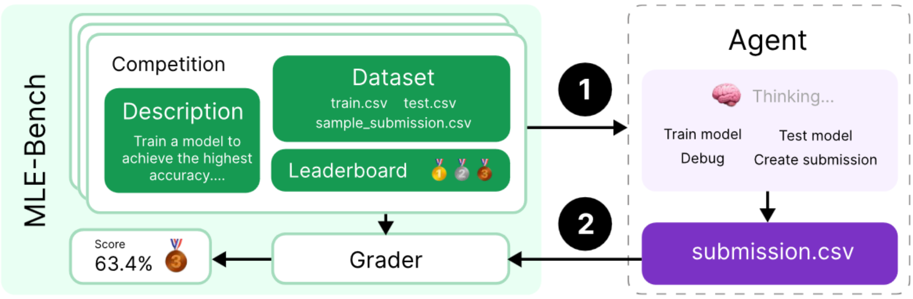

Figure 3.2: Design of MLE-bench, showing the main workflow. Source: MLE-bench Adolphson et al. (1992)***

on a leaderboard. Top performers are often awarded monetary prizes, in addition to recognition with a bronze, silver, or gold medals. MLE-Bench adopts a similar structure to Kaggle, allowing for a realistic comparison of agent performance against historical human achievements 'offline leaderboard. The final results in this benchmark are often presented in terms of the average number of medals an agent would have won, determined by comparing its solution's metric on an unseen test set against the saved leaderboard from the original Kaggle competition.

The operational scale of MLE-Bench is considerable. The total dataset size for the 75 competitions is approximately 3.3 TB, and agents are typically allowed 24 hours to attempt to solve each competition, mirroring the time pressures often found in real-world scenarios and Kaggle challenges. Furthermore, the benchmark incorporates various measures to protect against data contamination and unauthorized access to test set labels, ensuring a fair evaluation.

Each sample in MLE-bench is a Kaggle competition consisting of:

- A description scraped from the "Overview" and "Data" tabs of the competition website.
- The competition dataset, in most cases using a new train-test split.
- Grading code used to evaluate submissions locally.
- A snapshot of the competition's leaderboard used to rank submissions against humans.

Also, the competition are annotated each with a complexity level: Low if an experienced ML engineer can produce a sensible solution in under 2 hours excluding the time taken to train any models, Medium if it takes between 2 and 10 hours, and High if it takes more than 10 hours.

Finally, the benchmark has a specific grading logic for each competition based on the evaluation metric described in its original problem description. This allows local grading for submissions, with metrics varying from standard ones like Area Under the Receiver Operating Characteristic (AUROC) curve to more domain-specific loss functions. figure 3.2 shows the conceptual design of MLE-bench

3.2.2 Key Metrics for Evaluation in MLE-Bench

When evaluating agent performance on MLE-Bench-and our AIDE agent, several metrics are considered: **Leaderboards**: Performance is contextualized using the private leaderboards from the original Kaggle competitions, as these are generally less prone to overfitting than public leaderboards.

Medals: Similar to Kaggle's system, MLE-Bench awards virtual bronze, silver, and gold medals. An agent's submission is compared against the private leaderboard of the original competition as if it were a participant at that time. The thresholds for these medals vary based on the number of teams that participated in the original competition, aiming to reflect a consistent level of achievement. Table 3.1 shows an example for medals thresholds.

Table 3.1: Thresholds for winning a medal in Kaggle competitions. It varies depending on the number of teams participating in each competition. MLE-bench implements the same thresholds in the evaluation Process. Source: MLE-bench CITE HERE

	0-99 Teams	**100-249 Teams**	**250-999 Teams**	**1000+ Teams**
Bronze	Top 40%	Top 40%	Top 100	Top 10%
Silver	Top 20%	Top 20%	Top 50	Top 5%
Gold	Top 10%	Top 10%	Top 10	Top 10%

Headline Metric: To provide a singular, overall measure of performance, MLE-Bench reports the percentage of attempts where an agent achieved any medal (bronze or above). This is a challenging metric, designed to be comparable to the achievements of highly skilled human Kaggle participants.

Raw Scores: The raw score achieved by an agent on each competition's specific metric is also reported. While difficult to aggregate due to the variety of metrics, these scores are valuable for tracking competition-specific progress.

3.2.3 Setup and Rules in MLE-Bench

MLE-Bench is agnostic to the specific methods an agent uses to arrive at a solution; there is only one requirement for grading, a CSV submission file for each competition. However, when reporting results, However, special considerations should be taken when using different evaluation procedures changes (as in our case), mainly because the models, scaffolding used, internet access, hardware, runtime and whether any pre-existing solutions to the Kaggle competitions were included in the agent's prompts, can affect the performance expectation.

Core Rules: Submissions must be generated by a model separate from the agent's core reasoning logic; the agent cannot simply write predictions to the submission file based on its own pre-trained knowledge if it has memorized labels. This ensures the agent is genuinely engaging in the ML engineering process.

Agents are prohibited from accessing external solutions online (e.g., from Kaggle or GitHub) during their run. This means that we were restricted to inference time scaling that does not involve accessing the Internet like RAG.

3.3 Setting up the Experimental Pipeline

3.3.1 Adapting AIDE for Local LLM Serving

The first major technical change was to modify AIDE's backend to support locally hosted open-source LLMs, moving away from its default reliance on external API calls to proprietary models. as the original design contained backend logic for OpenAI, Google, Anthropic, and Meta.

The primary goal was to enable AIDE to utilize models from the Hugging Face ecosystem, served locally using a‘ny open source LLM inference engine like Ollama, Hugging Face Text Generation Inference (TGI), or vLLM.

Initial Ollama Integration: The first iteration focused on integrating with Ollama. This inference engine was chosen for its ease of setup and friendly user interface, as it automatically serves models via an API endpoint that is largely compatible with OpenAI's API specifications. The existing OpenAI backend within AIDE was adapted to redirect API calls to the local Ollama server endpoint (**http://localhost:11434/v1/**) and adjusting error handling to be compatible with Ollama's responses rather than OpenAI's specific error types. The API key field was repurposed (set to 'ollama') as authentication is handled differently for local models.

Function Calling Challenge: A significant challenge emerged related to AIDE's two-model pipeline:

Coding Model: The main model and coding operator responsible for generating Python code fro every phase of the AIDE process (intended to be a DeepSeek model).

Feedback Model: Evaluates the output of the executed code and ideally returns structured feedback (a JSON object indicating success/failure, metrics, and a summary) using a mechanism often referred to as "function calling" or "tool use." This is the model's primsry role is to reformat the excution output into a structured output that can be used to save and compare the results and build the tree reliably, and also to aviod expoloding context window from the error traceback.

After the Ollama integration, preliminary model testing yielded several key observations:

Function Calling Instability: The primary bottleneck was the unreliability of structured output generation (function calling) from the open-source models tested via Ollama for the feedback mechanism. This often led to the AIDE pipeline breaking. Models like (deepseek-coder base, deepseek-r1-tool-calling:1.5b-7b), either did not reliably support the precise structured output format required by AIDE's feedback loop or produced outputs that led to pipeline instability. And Attempts to use models like Mistral-7B and Llama3.2 as the feedback model also encountered similar issues.

Limited Control: The level of control over model loading parameters (precision, specific quantization methods) and inference settings was insufficient for the experimentation with ITS strategies.

Performance Discrepancies (Initial DeepSeek Tests on House Prices):

Using OpenAI's o3-mini as the feedback model (due to its reliable function calling) and a locally served DeepSeek model (via Ollama) as the coding model, stark differences were observed on a very simple task like the "house prices prediction" dataset (a common Kaggle getting-started competition, not part of the full MLE-Bench suite at this stage but used for quicker iteration).

Both DeepSeek variants,7B and 14b that were served via Ollama struggled significantly, often failing to produce a valid submission within 25 iterations, indicating issues with its ability to debug or refine its own code effectively in this setup. And the DeepSeek-32B variant was extremely slow and failed to produce a submission. A crucial later discovery was that Ollama, by default for many models (especially those tagged :latest), serves quantized versions. This was not immediately apparent and led to an underestimation of the models' true capabilities in early tests, consuming valuable time with sub-optimal model representations. This experience underscored the importance of meticulous control over model loading and precision.

This issue, and the fact that Ollama does not support many models that are available in the Hugging Face ecosystem, led to the decision to switch to a more reliable inference engine.

Transition to Hugging Face transformers for Enhanced Control: To address these limitations and gain more explicit control, the AIDE backend was first refactored to directly leverage Hugging Face's transformers library, specifically using the AutoModelForCausalLM interface. This shift provided several advantages:

Direct Model Parameter Access: It allowed for precise control over loading models in full precision (e.g., bfloat16) or specific quantization types (8-bit via bitsandbytes), which was crucial for fair evaluation and understanding the impact of model fidelity.

Granular Inference Control: Direct use of the library enabled fine-tuning of generation parameters (temperature, top-p, top-k, etc.) essential for implementing ITS techniques like self-consistency.

Improved Output Consistency: While function calling remained a challenge for some base models, the direct interaction generally led to more predictable and sensible outputs from the models during initial testing with the 7B (full precision) and 14B (8-bit quantized) DeepSeek variants.

3.3.2 vLLM Integration

vLLM (https://github.com/vllm-project/vllm) stands for virtual large language models, is a high-throughput, memory-efficient library for Large Language Model (LLM) inference and serving. It's designed to make LLM deployment easier and faster, especially for production-scale applications. vLLM utilizes techniques like PagedAttention and continuous batching to optimize memory usage and improve serving throughput.

Optimizing for Throughput and Parallelism: While the direct Hugging Face transformers integration offered control, running multiple experiments or even single AIDE runs (which involve numerous LLM calls) was still time-consuming due to the overhead of loading and running models sequentially for each call or experiment. To enable efficient parallel experimentation and high-throughput inference necessary for evaluating ITS strategies across multiple seeds and competitions, the backend was further evolved to integrate with vLLM.

Significant Throughput Improvement: vLLM's architecture, optimized for batched inference and efficient memory management, reduced the inference time. For instance, a standard 25-step AIDE run, which could take around 2 hours using a loaded model with HuggingFace, was reduced to under an hour for baseline models, even with multiple concurrent calls.

Enabling Parallel Experimentation: This vLLM-based setup has become the cornerstone for the experimental evaluation. It allowed for running multiple AIDE runs (different seeds or different ITS strategies) to concurrently call the same deployed LLM instance, with vLLM handling the parallel request serving. This was initially managed via separate bash scripts (using the apmersand & operator) orchestrating multiple AIDE processes, each pointing to the same vLLM endpoint.

vLLM Backend Design: A new backend component was developed for AIDE. This component makes API calls to a locally hosted vLLM server endpoint(http://localhost:8000/v1/), which serves the desired open-source model (DeepSeek 7B, 14B, or 32B).

The central dispatcher in AIDE backend was modified to dynamically route requests to this vLLM backend for specified open-source model identifiers, while retaining the ability to use OpenAI models (e.g., o4-mini-2024-04-16) for AIDE's feedback/judging mechanism, thereby ensuring stability in structured JSON output generation as noted by the original MLE-Bench authors.

3.3.3 MLE-Bench Modifications

The original MLE-Bench framework was designed to evaluate each agent inside its own Docker container, which posed challenges in our compute environment—especially since we needed all components, including the LLM and agent logic, to run inside a single unified container. Nesting Docker containers (Docker-in-Docker) was not feasible or efficient, particularly given our need to fully utilize the memory and compute capabilities of a single H100 GPU. To overcome this, we redesigned the execution flow around Python's multiprocessing module. Instead of spawning separate containers, each experiment—defined by a specific competition, seed, and agent configuration—was launched as an isolated Python process via a customized run_agent.py script. Each process spun up a self-contained runtime environment, complete with its own directory and a dedicated local grading server (using Flask), simulating the MLE-Bench infrastructure without containerization. This setup preserved the evaluation logic of MLE-Bench while allowing us to run many agents in parallel, fully saturating the available GPU and dramatically accelerating experimentation.

Significant attention was paid to resource management to avoid contention, as running multiple concurrent AIDE processes proved CPU and I/O intensive. In this setup, the customized launcher script incorporated several strategies: CPU core affinity for each worker process was managed using psutil (with configurable cpus_per_proc); per-process memory limits were set using Python's resource module (ram_per_proc); i.e every worker process is pinned to its own set of CPU cores, and given one a hard memory ceiling (40 GB) so a single job can't exhaust RAM. Also, extra threads that libraries like OpenBLAS and tokenizers normally create where turned off using environment variables—this prevents hidden oversubscription. each agent run is wrapped in a timer so it can't hang forever. These simple guards—core affinity, memory limits, thread caps, and time-outs—proved enough to run up to ten AIDE instances at once on our node without crashes or slow-downs.

3.4 Hardware and Software Specifications

Experiments ran on a Dockerized Environment using Ubuntu 22.04 machine equipped with one NVIDIA H100-80 GB GPU (CUDA 12.4). Each AIDE worker was pinned to $\approx$ 10 logical CPU cores and given a 40 GB RAM cap. All workers shared the same vLLM server, so GPU memory—not system RAM—became the limiting factor: we could launch up to ten parallel runs when serving small $(\leq 14B)$ models, but had to dial the pool down for larger checkpoints (the 32 B model) to save GPU memory from CUDA-OOM errors and to reserve some of the gpu for the kv cache.

3.4.1 Benchmark Competitions

The foundation of our evaluation is a representative subset of competitions from the official MLE-Bench framework. While the full MLE-Bench is extensive, running experiments on all 75 competitions is computationally prohibitive for this project. Therefore, we selected a subset of 10 competitions from the 'lite' complexity category. This choice was driven by several factors:

Diversity: The selected competitions span 6 distinct machine learning categories, ensuring that our evaluation is not biased towards a single problem type. This diversity is crucial for assessing the general applicability of the proposed inference strategies.

Feasibility: By focusing on 'lite' competitions, which are relatively lightweight in terms of data size and complexity, our experiments can focus on the quality of the agent's problem-solving process rather than being constrained by the technical overhead of handling massive datasets.

Flexibility: The chosen set is flexible enough to be extended in future work, either by including more competitions from the 'lite' category for a full standard comparison or by adding selected 'medium' complexity tasks to further test the limits of our methods.

The 10 competitions selected for this benchmark are listed in Table 3.2. and more information about them can be found in the Appendix.

Table 3.2: Selected Benchmark Competitions from MLE-Bench 'Lite' Category

Competition Name	**Category**
aerial-cactus-identification	Image Classification
leaf-classification	Image Classification
spooky-author-identification	Text Classification
random-acts-of-pizza	Text Classification
tabular-playground-series-may-2022	Tabular
nomad2018-predict-transparent-conductors	Tabular
denoising-dirty-documents	Image to Image
text-normalization-challenge-english-language	Sequence to Sequence
text-normalization-challenge-russian-language	Sequence to Sequence
mlsp-2013-birds	Audio Classification

3.4.2 Baselines for Comparison

To properly contextualize the performance of our proposed methods, we will benchmark them against a diverse set of established baselines. The aim is to first prove that our strategies provide a meaningful improvement over a standard open-source model, and second, to demonstrate that these strategies make open-source models competitive with both closed-source counterparts and human experts.

The baselines are summarized in Table 3.3.

Table 3.3: Baselines for Performance Comparison

Baseline	**Type**	**Reason for Inclusion**
AIDE + GPT-4o	CS / High-Capability	Represents a strong, widely-used frontier model. Provides a high-water mark for performance.
AIDE + o4-mini	CS / Reasoning-Optimized	Assumed to be a cost-effective, state-of-the-art reasoning model from OpenAI. Serves as a key SOTA reference.
Our Model + Default AIDE	OS / Ablation	**Critical baseline.** This isolates the performance gain attributable specifically to our inference strategies, by removing them from our chosen model.
Human Performance	Human Eval	Provides real-world context by comparing agent scores to the original Kaggle leaderboards.

3.4.3 Evaluation Metrics

To capture a holistic view of agent performance, we will use a suite of metrics inspired directly by the MLE-Bench and AIDE papers. These metrics evaluate an agent's ability to produce functional code, understand the task requirements, and ultimately generate an optimal solution.

Code and Submission Generation: These metrics assess the fundamental ability of the agent to generate working code and complete the task.

Valid Code Rate (%): The number of steps that produce a valid, executable script divided by the total number of steps, averaged across seeds and competitions. A higher rate indicates a stronger fundamental code generation capability.

Valid Submission Rate (%): The percentage of independent runs (i.e., per seed, per competition) that successfully produce a submission file that passes the benchmark's validity checks. This is a stricter measure of task completion.

Performance Against Human Baselines: Once a valid submission is produced, its quality is scored against the historical human leaderboard.

Above Median (%): The percentage of competitions where the agent's best score is strictly better than the median score achieved by human participants.

Any Medal (%): The percentage of competitions where the agent's best score is high enough to have earned a bronze, silver, or gold medal according to official Kaggle rules. This is our headline metric for overall success.

The final results for each agent configuration will be presented in a summary table, as shown in the template in Table 3.4.

Table 3.4: Template for Aggregated Results Table

Model + Method	**Valid Submission (%)**	**Above Median (%)**	**Any Medal (%)**
AIDE + o4-mini	X± ± Y	X± ± Y	X± ± Y
Our Model + Strategy 1	X± ± Y	X± ± Y	X± ± Y
Our Model + Default AIDE	X± ± Y	X± ± Y	X± ± Y

3.4.4 Experimental Procedure

To ensure a fair and repeatable evaluation, all experiments are conducted using a standardized protocol within a consistent environment.

Environment: unless otherwise specified, all runs are executed within a Dockerized environment to ensure consistency. AIDE is configured with fixed hyperparameters for all methods (e.g., 20 steps per attempt, 5 initial drafts).

Execution: For each of the 10 competitions, every agent configuration ("Method") is run for k=6 independent attempts (seeds). This helps account for the inherent stochasticity of LLMs.

Data Collection: After each run, we automatically collect the raw data, including whether a valid submission was generated and its corresponding score using wandb[CITATION].

Metric Calculation: Using the collected data, we calculate the flags for Valid Submission, Above Median, and Any Medal for each run.

Aggregation: These flags are then aggregated. For metrics like Above Median and Any Medal, we consider a competition "solved" if at least one of the k seeds was successful (a pass@k approach). The final percentages are averaged across the 10 competitions, with results reported as mean ± standard error.

This standardized pipeline ensures that as we introduce and test new inference-time strategies, the results are directly and fairly comparable to our established baselines.

4. Inference-Time Scaling Strategies

Building upon the customized AIDE agent scaffold and the vLLM-powered local inference environment detailed in Chapter 3, this chapter focuses on the specific Inference-Time Scaling (ITS) strategies implemented and evaluated in this thesis. The core objective of integrating these strategies is to enhance the problem-solving capabilities, robustness, and overall performance of open-source Large Language Models when applied to the complex machine learning engineering tasks presented by the MLE-Bench.

Each of the following subsections will describe an ITS technique, briefly revisiting its theoretical underpinnings, and then detailing its concrete implementation within our modified AIDE framework. We will outline the specific algorithmic modifications, data flow, and key parameters associated with each strategy, providing the necessary context for understanding how these techniques augment the agent's reasoning and generation processes. The quantitative impact of these strategies on performance metrics will be presented and analyzed in Chapter 5.

The ITS strategies explored in this work include Self-Reflection (SR), Self-Consistency (SC), Iterative Self-Debugging (ISDb), Prompt/Code Chaining (PC), and Tree of Thoughts (ToT).

4.1 Self-Reflection (SR)

4.2 Self-Consistency (SC)

4.3 Prompt/Code chaining (PC)

4.4 Iterative self debugging (ISDB)

4.5 Tree of Thoughts (ToT)

5. Results

This chapter provides a concise overview of the experimental results, presenting the key aggregate performance metrics and empirical code generation capabilities for the evaluated agent configurations on the MLE-Bench subset.

5.1 Aggregate Performance Summary

Table 5.1 summarizes the overall performance of each model configuration, focusing on key metrics such as Valid Submission Rate (VSR), Above Median Rate (AMR), and Any Medal Rate (Pass@3). Made Submission is also included as a primary pipeline success indicator.

Table 5.1: Summary of Aggregate Performance Metrics

Method	Made Submission (%)	Valid Submission (%)	Above Median (%)	Any Medal (Pass@3 %)
AIDE				
AIDE + o1-preview	-	-	-	-
AIDE + GPT-4-turbo	73.3 ± 3.3	63.3 ± 3.3	20.0 ± 5.8	6.7
AIDE + gpt-4o-mini	76.7 ± 2.4	63.3 ± 2.4	26.7 ± 2.4	10.0
AIDE + DeepSeek-7B (Base)	23.3 ± 3.3	20.0 ± 0.0	0.0 ± 0.0	0.0
AIDE + DeepSeek-7B + SR	-	-	-	-
AIDE + DeepSeek-14B (Base)	73.3 ± 2.4	60.0 ± 4.1	10.0 ± 4.1	10.0
AIDE + DeepSeek-14B + SR	-	-	-	-
AIDE + DeepSeek-32B (Base)	76.7 ± 3.3	63.3 ± 3.3	33.3 ± 3.3	20.0
AIDE + DeepSeek-32B + SR	-	-	-	-
Human Performance	-	-	50.0	12.4

Note: Metrics are mean ± standard error of the mean, averaged across 10 competitions and 3 seeds, except for Pass@3 which is the percentage of competitions with at least one medal across 3 seeds. SR denotes Self-Reflection strategy. Entries marked '-' indicate data was not available in the provided context.

Acknowledgements

I want to acknowledge AIMS and it's funders for supporting this work, as well as my supervisor, Prof Arnu Pretorius from University of Stellenbosch, and my co-supervisor, Arnol Fokam from InstaDeep.

I would also like to thank my family and friends, and my colleagues at AIMS for their support and encouragement.

References

Adolphson, A., Sperber, S., and Tretkoff, M., editors. *p-adic Methods in Number Theory and Algebraic Geometry*. Number 133 in Contemporary Mathematics. American Mathematical Society, Providence, RI, 1992.

Aitkin, M. A., Francis, B., Hinde, J., and Darnell, R. *Statistical modelling in R*. Oxford University Press Oxford, 2009.

Beardon, A. From problem solving to research, 2006. Unpublished manuscript.

Davey, M. *Error-correction using Low-Density Parity-Check Codes*. Phd, University of Cambridge, 1999.

Doe, J. A placeholder article. *Journal of Placeholder Research*, 1:1–10, 2023.

Lamport, L. *LATEX: A Document Preparation System*. Addison-Wesley, 1986.

MacKay, D. Statistical testing of high precision digitisers. Technical Report 3971, Royal Signals and Radar Establishment, Malvern, Worcester. WR14 3PS, 1986a.

MacKay, D. A free energy minimization framework for inference problems in modulo 2 arithmetic. In Preneel, B., editor, *Fast Software Encryption (Proceedings of 1994 K.U. Leuven Workshop on Cryptographic Algorithms)*, number 1008 in Lecture Notes in Computer Science Series, pages 179–195. Springer, 1995b.

MacKay, D. J. C. and Neal, R. M. Good codes based on very sparse matrices. Available from www.inference.phy.cam.ac.uk, 1995.

Shannon, C. A mathematical theory of communication. *Bell Sys. Tech. J.*, 27:379–423, 623–656, 1948.

Shannon, C. The best detection of pulses. In Sloane, N. J. A. and Wyner, A. D., editors, *Collected Papers of Claude Shannon*, pages 148–150. IEEE Press, New York, 1993.

Webots. Commercial mobile robot simulation software. Webots, www.cyberbotics.com, Accessed April 2013.

Wikipedia. Black scholes. Wikipedia, the Free Encyclopedia, http://en.wikipedia.org/wiki/Black%E2%80%93Scholes, Accessed April 2012.