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: Arnol Fokam, Arnu Pretorius
InstaDeep and Stellenbosch University, South Africa

12 June 2025
Submitted in partial fulfillment of an AI for science masters degree at AIMS South Africa

Abstract

Recent findings suggest that reasoning strategies applied during inference can bring greater performance gains in large language models (LLMs) than simply increasing model size. However, their application to open-source LLMs in complex Machine Learning (ML) engineering contexts remains largely unexplored.

In this work, we systematically implement and evaluate inference-time scaling (ITS) strategies within an open-source, agentic framework tailored to machine-learning engineering tasks. We enhance the AI-Driven Exploration (AIDE) agent scaffold with multiple ITS techniques—namely self-consistency, self-reflection, and modular task decomposition—and apply these strategies to distilled variants of DeepSeek-R1 models at parameter scales of 7B, 14B, and 32B. We evaluate the performance of our agents on a curated subset of competitions from the MLE-Bench, a comprehensive benchmark designed specifically to assess the machine learning engineering capabilities of AI agents.

Our experiments show that the best-performing agent—a 32 B DeepSeek model augmented with a decomposed planner-coder strategy—achieves a 30 % medal rate (pass@6), matching the performance of OpenAI o4-mini and surpassing the GPT-4-Turbo baseline.

Among the ITS methods tested, self-consistency and modular task decomposition proved particularly effective, while self-reflection provides the most benefit only to mid-sized models that are competent but flawed.

Our primary contributions are (i) an accessible, open-source, [1] specialized agentic scaffold optimized for ITS strategies and (ii) comprehensive empirical validation showing that carefully tailored inference-time strategies can enhance the capabilities of open-source LLMs.

Declaration

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

Asim Osman, 12 June 2025

[1] *aide-agent* GitHub repository

Contents

1. Introduction

1.1 From Model Size Scaling to Inference-Time Scaling (ITS)

The past few years have witnessed remarkable advancements in large language models (LLMs), driven primarily by scaling parameter counts, dataset sizes, and computational resources. Proprietary models from OpenAI (GPT series) (Brown et al., 2020), Meta (LLaMA) (Touvron et al., 2023), and Anthropic (Anthropic, 2023) exemplify this trend, showcasing unprecedented capabilities in language understanding, mathematical reasoning, and code generation. Although successful, this scaling paradigm relies on extensive pre-training that entails substantial computational costs and consumes increasingly scarce high-quality data (Bommasani et al., 2021). Even when no further parameter updates are required—as with inference-time strategies—access to these highly scaled models is often gated by proprietary APIs, usage fees, or restrictive licenses. Consequently, the combination of high pre-training costs and limited accessibility still *presents significant barriers* to the widespread use of state-of-the-art LLMs in machine-learning engineering, especially in resource-constrained settings.

Recently, research attention has shifted toward *inference-time* (or test-time) scaling techniques. Rather than investing in costly pre-training, inference-time scaling enhances existing models by dedicating additional computational resources during prediction. These methods allow a model to "think longer" about challenging problems by dynamically generating multiple reasoning paths (Wang et al., 2022), employing voting mechanisms (Chen et al., 2022), or iteratively refining its output (Madaan et al., 2023). Studies such as DeepMind's compute-optimal scaling work (Snell et al., 2024a) show that smaller models can achieve impressive performance through adaptive inference strategies, sometimes even outperforming substantially larger models. This paradigm has been validated extensively in the mathematical-reasoning community: techniques like self-consistency (Wang et al., 2022), self-reflection (Renze and Guven, 2024), and tree search (Yao et al., 2023a) routinely boost small models to near state-of-the-art accuracy on benchmarks such as GSM8K (Cobbe et al., 2021) and MATH (Hendrycks et al., 2021; Ehrlich et al., 2025; Li and Wu, 2024).

1.2 Problem Statement

Despite their success in mathematical reasoning, ITS strategies remain under-explored in the domain of machine-learning (ML) engineering. In the coding domain, correctness cannot be verified by simple string matching; instead, candidate programs must be executed against tests (Liu et al., 2023), a procedure that demands an automated, agentic framework. Concurrently, the emergence of powerful open-source "reasoning" models—such as the distilled variants of DeepSeek's R1 series 7 B, 14 B, and 32 B (Guo et al., 2025a)—provides an ideal test-bed for investigating these techniques. These models possess strong reasoning abilities yet are compact enough to run on a single GPU.

The central problem addressed in this thesis is the systematic evaluation of ITS strategies on accessible, open-source models for complex, end-to-end ML-engineering tasks. We contend that ideas proven effective for mathematical reasoning can transfer to ML engineering, *provided they are embedded in an agent that can autonomously run, debug, and improve its code*, thereby converting raw inference-time compute into an effective, automated engineering workflow.

MLE-Bench (Chan et al., 2024) offers an ideal benchmark for this purpose. Designed to assess the ML-engineering capabilities of AI agents, MLE-Bench consists of 75 diverse Kaggle competitions and evaluates an agent's ability to design and implement complete ML pipelines. In this work we focus on

ten competitions spanning six contemporary ML-engineering tasks, ensuring that our evaluation is not biased toward any single problem family.

1.3 Research Questions

This research is guided by the following questions:

- **Q1.** To what extent can ITS strategies, adapted from mathematical reasoning, improve the performance of distilled DeepSeek models on complex, multi-step ML-engineering tasks?
- **Q2.** What are the performance trade-offs (e.g., benchmark score, computational cost, success rate) among different ITS strategies—Self-Consistency, Self-Reflection, and a hybrid Code-Chaining (task-decomposition) approach?
- **Q3.** How do these enhanced open-source models (7 B, 14 B, and 32 B) compare with established baselines, including proprietary models such as GPT-4-Turbo and OpenAI O4-mini, on the MLE-Bench benchmark?

1.4 Scope and Contributions

To answer these questions we integrate three complementary ITS techniques—Self-Consistency, Self-Reflection, and modular task decomposition—into "AIDE," an open-source agent scaffold for ML competitions (Jiang et al., 2025). Each technique is realised as a distinct agent variant and executed via the high-throughput `vLLM` inference engine (Kwon et al., 2023). Evaluation is conducted on a curated subset of MLE-Bench to enable rapid, systematic comparison of models and strategies.

Our contributions are threefold:

- **Infrastructure.** We release an extensible, open-source agent that couples AIDE with a high-throughput inference engine for locally hosted models and provides plug-and-play ITS modules that can be enabled dynamically.
- **Empirical Insight.** We present a controlled study of ITS strategies (Self-Reflection, Self-Consistency, and Task Decomposition) across three model sizes (7 B, 14 B, 32 B). The analysis reveals a clear relationship between model scale and strategy efficacy, identifying the optimal use cases for each technique.
- **Performance Milestone.** We demonstrate that a 32 B DeepSeek model, when augmented with our *Decomposed Task Generation* strategy, attains a 30 % medal rate (pass@6)—a **+10-point** improvement over the 32 B baseline, surpassing GPT-4-Turbo and matching the state-of-the-art O4-mini on our benchmark subset.

1.5 Thesis Outline

The remainder of this thesis is organised as follows. Chapter 2 reviews background and related work on LLMs, inference-time scaling, and agentic frameworks. Chapters 3 and 4 detail the methodology, including the experimental setup, the AIDE scaffold, MLE-Bench, and the implementation of our ITS agent variants. Chapter 5 presents quantitative results and qualitative analyses. Finally, Chapter 6 discusses the findings, outlines limitations, and suggests directions for future research.

2. Background and Related Work

This chapter provides an overview of the main research areas that form the foundation of this thesis. We begin by providing context for the role of LLMs in code generation and introducing the specific DeepSeek models (Guo et al., 2025a) used in our work. We then discuss the agentic frameworks that enable our experiments, namely the AIDE scaffold and the MLE-Bench benchmark. The core of this chapter is a detailed overview of the relevant ITS strategies, including the ones that we implement and evaluate. Finally, we synthesize this information to clearly define the research gap this thesis aims to address.

2.1 Large Language Models (LLMs)

LLMs (Zhao et al., 2023) are deep neural networks, based on the Transformer architecture (Vaswani et al., 2017), which are trained on extensive text and code datasets. By leveraging a straightforward predictive objective over enormous volumes of data, these models have demonstrated remarkable capabilities in a diverse range of natural language and programming tasks. Recent advancements, exemplified by models such as OpenAI's GPT series (Brown et al., 2020), Meta's LLaMA family (Touvron et al., 2023), and Anthropic's Claude (Anthropic, 2024), have been driven by significant scale, with models reaching hundreds of billions of parameters.

This substantial scale has unlocked "emergent" abilities and advanced skills like complex reasoning and in-context learning that are typically absent in smaller models (Wei et al., 2022). These capabilities enable LLMs to solve arithmetic problems, answer complex questions, and generate functionally correct code, often without explicit task-specific training. The pursuit of these abilities has been guided by scaling laws, which demonstrate that an LLM's performance, as measured by metrics like cross-entropy loss, improves in a predictable, power-law relationship with increases in model size, dataset size, and computational budget (Kaplan et al., 2020; Hoffmann et al., 2022).

These scaling laws provide reliable performance guarantees for achieving greater performance, and drove a competitive race for building larger models, leading to the current models with hundreds of billions of parameters. However, this train-time scaling approach faces fundamental limitations. The computational and financial costs associated with training increasingly large models are immense, and the industry is encountering a scarcity of high-quality training data relative to the demands of ever-larger models (Bommasani et al., 2021). These challenges have motivated a shift in focus from costly pre-training to more efficient methods of performance enhancement, creating a ground for exploration.

2.2 LLMs for Code Generation

The application of Large Language Models to programming tasks evolved from simple autocompletion to sophisticated, multi-step program synthesis. This evolution was first introduced by OpenAI's Codex, a model fine-tuned on 100 billion lines of public code, which provided the engine for the first version of GitHub Copilot (OpenAI, 2021). This demonstrated that LLMs could not only generate syntactically correct code but also capture complex programming patterns and idioms.

The capabilities of modern code LLMs are typically assessed on several benchmarks. Early benchmarks like HumanEval (Chen et al., 2021) and MBPP (Mostly Basic Python Programming) (Austin et al., 2021a) focus on "function-level" generation, where the model must complete a single Python function

from a docstring. While foundational, these benchmarks do not capture the complexity of real-world software development. More recently, the field has moved towards more realistic evaluations. Google's DeepMind introduced AlphaCode (Li et al., 2022b), which achieved an average rank in the top 54% on Codeforces, a competitive programming platform, demonstrating an ability to tackle unseen problems that require complex algorithmic reasoning (Li et al., 2022a).

Arguably one of the most challenging modern benchmarks is SWE-bench, which evaluates an LLM's ability to resolve real-world GitHub issues from popular Python repositories (Jemison et al., 2024). This difficult task requires models to understand large, unfamiliar codebases, formulate a plan, and perform precise, multi-file edits. On this benchmark, even state-of-the-art proprietary models like Claude 3 Opus achieve a success rate of only around 11% (Anthropic, 2024). The emergence of agentic systems like Devin, which claims a 13.86% success rate on SWE-bench, underscores that raw model intelligence is insufficient; it must be combined with sophisticated scaffolding that allows the agent to plan, use tools (like a code editor and shell), and debug (Cognition, 2024).

Despite these remarkable advancements, the most capable models and agents remain proprietary. This poses a significant barrier for controlled experimentation and research and motivates the research questions of this thesis: for open-source models, where we have complete control over how the information fed into these models is managed, to what extent can advanced inference strategies, begin to close the performance compared to proprietary models?

2.3 Reasoning-Distilled Models: DeepSeek-R1

The models used in this thesis are specifically chosen for their unique training methodology, which prioritizes reasoning ability through a process of reinforcement learning and subsequent distillation. They originate from the DeepSeek-R1, a powerful "teacher" model designed for complex reasoning (Guo et al., 2025a).

The teacher, DeepSeek-R1, is a 671B parameter Mixture-of-Experts (MoE) model, with 37B parameters being active during any given forward pass. Its development pipeline is multi-staged and designed to elicit and refine reasoning. The process begins with DeepSeek-R1-Zero, a model trained via pure reinforcement learning (RL) using the Group Relative Policy Optimization (GRPO) algorithm, without a preliminary supervised fine-tuning (SFT) step. While this "pure RL" approach successfully incentivizes powerful reasoning behaviors, it results in outputs with poor readability and language mixing.

The more refined DeepSeek-R1 pipeline involved:

1.Cold Start SFT: A base model is first fine-tuned on a small set of high-quality, long chain-of-thought (CoT) examples to create a more stable initial actor for RL.

2.Reasoning-Oriented RL: The model then undergoes large-scale RL, similar to R1-Zero, to enhance its core reasoning capabilities.

3.SFT Data Generation: Crucially, upon nearing convergence, the RL-tuned model is used to generate a new, larger dataset for supervised fine-tuning. This is done via rejection sampling, keeping only the correct and coherent reasoning trajectories.

4.Final RL Stage: The model is retrained on this new SFT data before undergoing a final RL stage to align it with human preferences for helpfulness and harmlessness.

The models used in this thesis are directly benefiting from this sophisticated process. They are smaller, dense models (1.5B,7B, 8B, 14B, 32B, 70B) created by distilling the reasoning capability of DeepSeek-

R1. This is achieved by taking open-source base models from the Qwen (Bai et al., 2023) and Llama (Touvron et al., 2023) families and fine-tuning them on a curated dataset of 800,000 reasoning traces generated by the DeepSeek-R1 teacher. This SFT-based distillation effectively transfers the reasoning processes of the massive teacher model to the smaller student models, without requiring them to undergo the complex RL pipeline themselves.

This unique combination of features makes the distilled DeepSeek models the ideal testbed for our research:

Distilled Reasoning: They inherit the structured, step-by-step thinking processes of a state-of-the-art reasoning model.

Accessibility: They are open-source and computationally efficient enough to be hosted and extensively evaluated within an academic research environment.

Untapped Potential for Code: Their strong performance on mathematical reasoning benchmarks like AIME and MATH-500 is documented, but their application to agentic, end-to-end ML engineering remains a novel area of research.

2.4 Inference-Time Scaling (ITS)

As an alternative to costly train-time scaling, ITS has emerged as a powerful method for enhancing model performance. The core principle of ITS is to leverage additional computational resources during inference to enable a model to explore, evaluate, and refine its outputs, effectively allowing it to spend more time solving a problem.

The core idea in ITS for code generation is to allow a model to generate and evaluate multiple solutions or reasoning steps before finalizing an answer, much like a programmer iteratively writes, tests, and debugs code. This can enable smaller models to achieve results on par with much larger models by compensating with clever search and reasoning.

2.4.1 Sampling Approaches

The most direct application of ITS is to generate multiple independent outputs from the model and select the best one. This approach leverages the stochastic nature of LLM decoding to explore different solution paths.

Best of N Sampling: This strategy involves generating N candidate solutions and then using an external evaluation function, or "judge," to select the best one (Li et al., 2022b). In the context of coding, this judge can be a set of unit tests: the solution that passes the most tests is chosen. This method is highly effective but depends on the availability of a reliable and automated evaluator.

Self-Consistency and Majority Voting: When an external judge is unavailable, Self-Consistency provides a powerful alternative (Wang et al., 2022). This technique, which builds on Chain-of-Thought (CoT) prompting, by sampling multiple diverse reasoning paths and selecting the most frequent answer. This is the general form of majority voting. The underlying intuition is that while there may be many valid ways to reason about a problem, they should all converge on the correct solution. If multiple, distinct lines of reasoning produce the same answer, confidence in that answer increases. This technique has been shown to significantly improve performance on reasoning tasks and directly inspires us to implement it in our work.

2.4.2 Iterative Self-Improvement (Self-Reflection and Debugging)

Beyond generating independent samples, a more sophisticated class of ITS techniques involves having the model iteratively improve a single solution in a feedback loop. This mimics the natural workflow of a human programmer: write, test, and debug.

Self-Debugging: This paradigm explicitly tasks the LLM with finding and fixing bugs in its own generated code. For example, Chen et al. (2023) introduced Self-Debugging, where an LLM is taught via examples to run the code (or at least simulate running it) and then asked to explain what the code is doing and identify any mistakes based on either runtime results or logical reasoning. Notably, their approach has the model look at execution results (error messages or failed tests) without human feedback, and then pinpoint the bug and fix it – effectively the model effectively learns to debug itself. This yielded state-of-the-art results on several coding tasks, improving accuracy by up to 12% on test-driven benchmarks by iteratively correcting errors.

Self-Reflection: A more general version of this concept is Self-Reflection, where the model is prompted to critique its own solution and then refine it (Madaan et al., 2023). The Reflexion framework, for example, formalizes this by having an agent generate a textual reflection on a failed attempt, which is then added to the agent's memory to guide its next attempt (Shinn et al., 2023) . By verbalizing what went wrong, Shinn et al. (2023) shows that the model is less likely to repeat the same mistake. This family of self-corrective techniques is promising for coding tasks, as the binary feedback from a compiler or (pass/fail) provides a clear and unambiguous signal for the model to learn from at inference time. Our own implementation in this thesis is a direct application of this critique-and-revise principle.

2.4.3 Search-Based Approaches

The previous techniques either generate multiple independent solutions or iteratively refine a single one. An advanced approach treats the generation process as a formal search problem, allowing the model to plan, explore, and backtrack through a space of possible solutions.

Tree-of-Thought (ToT): This framework explicitly generalizes Chain-of-Thought by structuring the reasoning process as a tree (Yao et al., 2023a). Instead of following a single, linear chain of reasoning, a ToT agent can explore multiple reasoning paths simultaneously. At each step, the model generates several candidate "thoughts" or next steps. These are then evaluated—either by the LLM itself (via prompted self-critique) or a simple heuristic—and the most promising branches are selected for further expansion. This allows the agent to perform a lookahead search, assess the viability of different paths, and backtrack from mistakes. This structured exploration has proven highly effective for problems where the solution requires planning or is not immediately obvious, with ToT dramatically outperforming standard CoT on tasks like the "Game of 24" mathematical puzzle (Yao et al., 2023a). Our TOT Agent is a direct implementation of this philosophy, though benchmarking it was beyond our compute and time constraints.

Advanced Search Frameworks ($S*$): Other research has explored even more sophisticated search algorithms. The S* framework, for example, combines parallel sampling with iterative debugging (Sun et al., 2024). It begins by generating multiple initial programs. Each of these programs is then treated as the root of a separate search, where the agent iteratively debugs it using feedback from code execution. The framework can even use the LLM to generate adaptive tests that differentiate between two candidate programs to determine which is more correct. Such advanced search methods demonstrate the significant performance gains that can be achieved by systematically exploring and refining the solution space.

2.4.4 Tool-Assisted and Modular Reasoning

A final category of ITS involves decomposing the problem-solving process into distinct modules or giving the agent access to external tools. This offloads specific cognitive tasks, allowing the model to focus on its core strengths.

Tool-Assisted Reasoning (ReAct): The ReAct framework enables an LLM to synergize reasoning with action by giving it access to external tools (Yao et al., 2023b). In this paradigm, the model interleaves the generation of textual thoughts (internal reasoning) with actions (API calls to a search engine or code interpreter). The output of the action is then fed back into the model's context, informing its next thought. This allows the model to gather information, test hypotheses, and overcome knowledge gaps within a single inference loop. For code generation, the most critical tool is the execution environment itself. By running code and observing the output or traceback, the agent receives grounded, unambiguous feedback that can be used to debug and refine its solution, a process central to our AIDE-based methodology.

Role Decomposition (Planner-Executor): A common and effective modular strategy that is very close to the ReAct framework is the planner-executor paradigm (Erdogan et al., 2025; Biju et al., 2025). In this approach, the problem is split into two phases. First, an LLM acting as a "planner" is prompted to create a high-level, step-by-step plan. Second, the same LLM (or a different one) acting as an "executor" is given this plan and tasked with implementing it in code. This separation of concerns reduces the cognitive load on the model at each step, as it can focus on logical planning and code implementation independently. This is the core principle behind our Planner Coder Agent and Code-Chaining Agent, which separates the creation of a "master plan" from the segmented generation of the final script.

2.5 Frameworks for Agentic ML Engineering

The ITS strategies described above are powerful but require a sophisticated framework to be applied effectively to complex, multi-step problems like ML engineering. Such problems require an agent to interact with an environment, execute code, analyze results, and iteratively refine its approach rather than single-shot code generation. This section discusses the conceptual foudations of the two major frameworks used in this thesis: the AIDE agentic scaffold (Jiang et al., 2025) and the MLE-Bench (Chan et al., 2024)evaluation benchmark.

2.5.1 AIDE (AI-Driven Exploration) and Agent Scaffolding

"Scaffolding" refers to the ecosystem of tools, prompts, and control logic built around an LLM to augment its capabilities, enabling it to perform tasks that are impossible with a single inference call (Zaharia et al., 2024). This can range from simple prompt templates to complex agentic loops that give the model memory, tools, and the ability to plan.

AIDE represents a state-of-the-art agentic scaffold specifically designed for ML engineering (Jiang et al., 2025). Instead of treating the task as a single, long-horizon problem, AIDE frames it as a code optimization problem. Its core innovation is to formulate the iterative trial-and-error process as a tree search through the space of possible code solutions.

As described by Jiang et al. (2025) the AIDE framework is composed of several key conceptual components:

Solution Tree: A data structure where each node is a complete, executable Python script (a potential

solution) and each edge represents an improvement or debugging attempt.

Coding Operator: The LLM itself, prompted to perform one of three atomic operations: drafting a new solution from scratch, debugging a script that produced an error, or improving a valid solution to increase its score.

Evaluator: A stateless function that executes a solution script and returns a scalar score (validation accuracy). This score guides the search.

Search Policy: A set of heuristics that selects the next node from the tree to expand, determining whether to draft, debug, or improve.

Summarization Operator: A mechanism to distill the history of the solution tree into a concise summary, preventing the LLM's context window from overflowing while still providing relevant information from past attempts.

By structuring the problem this way, AIDE provides scalable environment for autonomously exploring the vast solution space of ML engineering tasks. It is open-source, which makes it a great candidate for integrating and testing the custom ITS agent variants developed in this thesis.

2.5.2 MLE-Bench

Evaluating the true capability of an ML engineering agent requires a benchmark that reflects the holistic nature of the task. While benchmarks like HumanEval (Chen et al., 2021) are invaluable for measuring function-level code generation, they do not assess an agent's ability to accurately perform ML task, from data loading and preprocessing to model training and submission.

To address this, MLE-Bench was created as way to benchmark LLMs on 75 real-world Kaggle competitions (Chan et al., 2024). Its design is centered on two main choices: (1) selecting challenging and representative tasks that mirror contemporary ML engineering work. (2) enabling direct comparison of agent performance against human-level baselines.

For each competition, MLE-Bench provides:

- A detailed problem description.
- The complete dataset, often with a recreated train-test split.
- Local grading code that replicates the original competition's evaluation metric.
- A snapshot of the original Kaggle leaderboard.

This structure allows an agent's final submission to be scored and ranked against the thousands of human competitors who participated in the original competition, providing a grounded measure of its real-world efficacy. The primary "headline metric" for MLE-Bench is the percentage of competitions in which an agent achieves a bronze medal or higher, a difficult metric that requires surpassing a significant fraction of human participants. By using MLE-Bench, this thesis can move beyond synthetic tests and measure the practical impact of ITS strategies on complex tasks.

2.6 Gaps in the Literature

The literature shows that Inference-Time Scaling strategies are a powerful and compute-efficient means of enhancing the capabilities of LLMs, and sophisticated agentic frameworks like AIDE provide the means to apply them to complex, real-world problems.

However, a critical research gap exists at the intersection of these domains. As shown in the AIDE and MLE-Bench papers, research and state-of-the-art results have been overwhelmingly dominated by large, proprietary, closed-source models (Jiang et al., 2025; Chan et al., 2024). Applying these advanced ITS strategies to smaller, more accessible, open-source models—remains a significant open area of research and exploration.

3. Methodology

3.1 Overview of the Experimental Setup

In this chapter, we describe the experimental setup development and design choices that went into adapting, building and evaluating our methods, starting with the agent scaffold, the MLE-Bench benchmark and the implemented ITS strategies. The primary objective is to have a rigorous and replicable procedure for quantifying the impact of these techniques on complex machine learning engineering tasks. We begin by describing the core components of the experimental environment, including the system architecture, the custom-developed high-throughput inference backend, and the foundational models used. We then describe the practical application of the AI-Driven Exploration (AIDE) agentic scaffold within our setup. The central focus of the chapter is a detailed and comprehensive description and explanation of the ITS agent variants developed for this thesis. Finally, we specify the evaluation protocol, including the selection of benchmark tasks from MLE-Bench, the baselines for comparison, and the metrics used to measure performance.

3.2 Core Components and Environment

A robust experimental pipeline was engineered to support the high-throughput, parallelized evaluation required for this research. This meant significant modifications to the standard AIDE and MLE-Bench frameworks to enable the use of locally hosted open-source models at scale.

All experiments were conducted within a dockerized environment on a system equipped with an NVIDIA H100 80GB GPU. The primary engineering challenge was to adapt AIDE, which defaults to API calls for proprietary models, to a setup that allows for the controlled and efficient serving of local open-source models.

Initial integration attempts with the Ollama inference engine were undertaken due to its ease of use. However, these early tests revealed two critical limitations: (1) the limited support for open source models, and the limited options in terms of quantization precision (2) a lack of granular control over model loading parameters, such as quantization and precision, which led to inconsistent performance.

To overcome these challenges, a custom back-end was developed that uses vLLM, a high-throughput LLM serving library designed for efficient, batched inference (Kwon et al., 2023). The target open-source models were hosted on a local vLLM server, which exposed them through an OpenAI-compatible API endpoint. A new backend component was written for AIDE to route all coding and planning requests to this local server. This architecture provided two advantages:

High Throughput: vLLM's use of PagedAttention and continuous batching enabled the concurrent processing of many requests, drastically reducing the time needed for experiments involving many LLM calls.

Parallel Experimentation: The setup allowed multiple, independent AIDE agent processes to run in parallel, each making requests to the same shared vLLM server instance. This was essential for efficiently executing experiments across different models, ITS strategies, and random seeds.

To manage system resources during these parallel runs, each agent process was pinned to a specific set of CPU cores and given a hard memory ceiling (40 GB) to prevent contention.

The models selected for this study are the reasoning-distilled DeepSeek variants, introduced in Chapter 2. Specifically, we evaluate three models from this series to analyze the effect of scale on the efficacy of ITS strategies, **DeepSeek-R1-Distilled-Qwen-7B**, **DeepSeek-R1-Distilled-Qwen-14B** and **DeepSeek-R1-Distilled-Qwen-32B**.

Two of the three reasoning-distilled DeepSeek variants were served in dynamic FP8 form using the Red Hat-optimised checkpoints DeepSeek-R1-Distill-Qwen-14B/32B-FP8-dynamic hosted on Hugging Face (AI, 2025). Quantising weights and activations from FP16 to FP8 halves both disk footprint and GPU-memory requirements while leaving perplexity "within a few tenths of a point" of the full-precision baseline, according to the model cards. Because our experiments run on NVIDIA H100-80 GB GPUs, this choice also unlocks specialised FP8 Tensor-Core kernels in the Hopper architecture, delivering up to 50% higher throughput relative to FP16/BF16 inference (NVIDIA Corporation, 2023, 2024). In practice the smaller memory footprint allowed us to keep the KV-cache for a 32 B model resident on-device while batching requests from up to ten parallel AIDE processes, with no measurable accuracy loss—an efficiency unattainable with full-precision checkpoints on a single H100.

3.3 AIDE: AI-Driven Exploration in the Code Space

At the center of this work, and the most critical component in our experimental setup is the agent scaffold itself, For this research work, we are leveraging AI-Driven Exploration (AIDE), an open source agent scaffold, developed by WecoAi Jiang et al. (2025). AIDE 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 focusing 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 debugging or improving that solution.

AIDE is powered by an LLMs, typically a capable model like GPT-4 or Claude. These LLMs are responsible for proposing new code, debugging existing scripts, or suggesting an 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, but the scope and the level of this scaling are high-level. The implementation of AIDE is publicly available, which was a key factor in its selection for this thesis, as it allows integration of custom inference time scaling methods.

3.3.1 Core Components and Methodology of AIDE

Below is a breakdown of how AIDE operates as outlined in their original paper Jiang et al. (2025)

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'.

Evaluator (h): This is a stateless function that takes a solution script as input and returns a scalar score (e.g., validation accuracy or 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

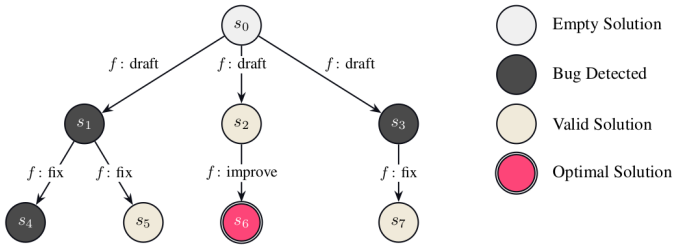


Figure 3.1: A sample solution tree for AIDE, illustrating the drafting, fixing, and improvement steps. (Source: Jiang et al. (2025), p. 4)

traceback) and produces a structured output via function calling. This particular role is considered stateless since it always produces the same score, and it does not consider 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 primarily based on this.

Search Policy (π): AIDE uses 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).
2. Debug a buggy node (if it's within a certain debug depth).
3. Improve an existing, non-buggy solution (typically targeting the current best-performing one).

This policy has practical heuristics, such as initially exploring diverse solutions and then iteratively refining the most promising ones, while also limiting attempts to fix persistently broken solutions. Although this might be the most obvious area of improvement in the AIDE pipeline, whether by using a learned model that guides this search, the nature of the problems that small open source models have is largely not caused by the search heuristic; instead, it is more fundamental in terms of ability to generate valid code in the first place, or correct usage of libraries and modules.

Coding Operator (F): This is the LLM itself as it proposes new scripts. It has three main entry points, each with specialized prompts:

1. **Drafting**: Create a completely new solution "exploration". The LLM is prompted to outline a plan to solve the problem (e.g., specify a neural network architecture and feature engineering ideas) and then generate a single-file Python program implementing that plan. These are then separated using regular expressions.
2. **Debugging**: Repairing buggy solutions by inspecting error logs and execution traces. It attempts to fix issues like broken imports or incorrect tensor dimensions while preserving the overall approach.
3. **Improving**: Called when a valid, non-buggy solution exists but could benefit from modifications (data preprocessing changes, architectural adjustments, or optimization tweaks and hyperparameter tuning). The LLM is prompted to propose a single "atomic" change, such as switching an optimizer or adding a regularization technique, so that its impact on performance can be directly measured.

Finally, to manage the LLM's context window and avoid "prompt explosion" from 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 execution output), This concise summary allows each code revision to be stateless yet guided by prior information, maintaining efficiency.

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.

The choice of AIDE as the foundational scaffold for this research is deliberate. It is open-source, and 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, serves as an ideal platform for experimentation.

AIDE has mainly been tested and evaluated only using lagre-scale proprietary 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.

3.4 MLE-Bench

To assess the performance of the enhanced aide scaffold we developed, we benchmark on the newly released MLE-Bench (Chan et al., 2024), a benchmark for measuring how well LLM-based agents perform at machine learning engineering, it is composed of 75 Kaggle competitions spanning a variety of domains, including natural language processing, computer vision, and signal processing. These competitions are curated from the Metakaggle dataset that contains around 5000 Kaggle competitions. ML competitions in MLE-Bench vary in category, scale, and complexity. This benchmark is designed not merely to test isolated skills, but to measure an agent's capacity for autonomous, end-to-end problem-solving—a workflow that spans model creation, training, evaluation, and iterative refinement. While MLE-Bench is a recent development, it is built upon a foundation of prior work aimed at evaluating LLMs coding and agentic abilities. Benchmarks like APPS (Hendrycks et al., 2021) and early evaluations of models such as Codex Chen et al. (2021) have been used to measure general coding competence. The choice of MLE-Bench, and its particular relevance to this research, lies in its focus on tasks that are both representative of modern industry practices and known to be challenging for current AI systems.

3.4.1 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. According to the authors, this benchmark focuses on two design choices:

- Selecting tasks that are challenging and representative of contemporary ML engineering work

- 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 on a leaderboard. Top performers are often awarded monetary prizes, in addition to recognition with 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 size for datasets in 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, to ensuring 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.

3.4.2 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.

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.

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 Chan et al. (2024)

	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 1%

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.

3.4.3 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, special considerations should be taken when using different evaluation procedures changes (as in our case). The reason for this is 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 Retrieval-Augmented Generation (RAG).

There are no specific limitations or rules on the time allowed for the agent run, the number of steps or compute infrastructure. and the competitions used for the evaluation, as the benchmark specifically designs a "lite" subset for comparisons, making our choice of evaluating on 10 competitions a valid evaluation.

3.5 Implemented Inference-Time Scaling (ITS) Agents

Building upon the AIDE scaffold and our custom inference environment, we developed and implemented several distinct agent variants, each working with a specific ITS strategy. These agents were created by extending a base Agent class and overriding its core generative operations to introduce more sophisticated reasoning and solution-finding mechanisms. The following sections detail the methodology behind each of these agents.

3.5.1 Self-Reflection

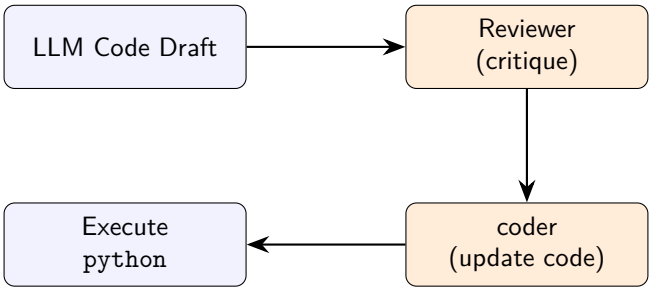


Figure 3.2: Two-step Self-Reflection: a draft is critiqued and updated *before* the first execution.

Initial baseline experiments with the standard AIDE agent revealed a significant performance gap between the smaller open-source DeepSeek models and their larger, proprietary counterparts. Qualitative analysis of the execution logs showed that the 7B in particular, frequently produced buggy initial code drafts. Furthermore, the model struggled to effectively utilize the feedback provided by the evaluation module during the standard debugging cycle, often failing to correct the root cause of an error or introducing new bugs.

To address this, the first ITS strategy we implemented was Self-Reflection (SR). This approach is conceptually motivated by frameworks like Reflexion (Shinn et al., 2023) and Self-Refine (Madaan et al., 2023), which demonstrate that an LLM's performance can be improved by prompting it to critically evaluate and iteratively revise its own output. The central hypothesis was that by introducing an explicit, structured reflection step before code execution, the agent could possibly identify and rectify its own errors more effectively than through the standard feedback loop alone.

Algorithmic Implementation

A two-step, self-reflection process was integrated into the AIDE agent's main operational loop. This mechanism is triggered during the drafting phase, a generated script is fed back to itself for checking. The same base coding LLM that generated the initial script is used for both stages of the reflection process, acting first as a "Reviewer" and then as a "Coder." The workflow is detailed in Algorithm 1.

Critique Stage (The "Reviewer" role): When a script is generated, the LLM is first prompted to act as a code reviewer. It is provided with the script and the high-level task description for context. The prompt explicitly instructs the model to identify 1-4 small mistakes (e.g., typos, simple logic errors, incorrect file paths) and to output only a natural language response. This response must consist of a brief summary of the main error, followed by a numbered list of concise, text-based instructions on how to fix it. The model is strictly forbidden from generating any Python code in this stage.

Revision Stage (The "Coder" role): The textual "fix instructions" generated by the Reviewer are then

passed to the second stage. The LLM is now prompted to act as a precise code editor. It receives the original buggy script alongside the fix instructions it previously generated. Its task is to apply only these minimal textual edits to produce a revised, complete Python script. The prompt strongly reinforces that the model should not change any other part of the code and must ignore any accidental code snippets that may have appeared in the instructions.

Evaluation: The revised script produced by the Coder then replaces the original code for the current step and is immediately passed back to the standard AIDE evaluation module. The outcome of this re-evaluation determines the final status of the node (e.g., buggy or not, performance metric) in the AIDE solution tree.

Algorithm 1 Two-Step Self-Reflection via Critique and Revision

Input: New Draft Code C_{draft}, Task Description D
Output: Revised Code $C_{revised}$
▷ — Stage 1: Critique Generation (Reviewer role) —
critique_prompt ← FormatReviewerPrompt(C_{draft}, D)
fix_instructions ← LLM.GenerateResponse(critique_prompt) ▷ LLM returns text-only instructions
if fix_instructions contains "No specific errors found" **then**
    return C_{draft} ▷ Return original code if no fix is proposed
▷ — Stage 2: Code Revision (Coder role) —
revision_prompt ← FormatCoderPrompt(C_{draft}, fix_instructions)
$C_{revised}$ ← LLM.GenerateCode(revision_prompt) ▷ LLM returns full revised script
return $C_{revised}$

3.5.2 Modular Code Generation (CodeChain or Decomposed Planner-Coder)

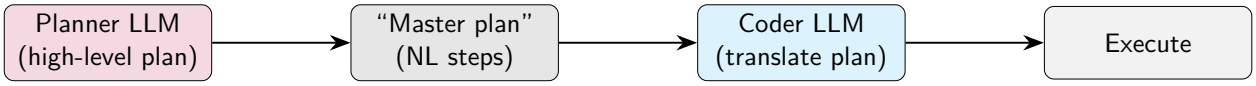


Figure 3.3: Planner–Coder task decomposition: strategy first, implementation second.

The second ITS strategy implemented explores task decomposition, a method designed to address the inherent difficulty of generating complex, end-to-end ML solutions in a single, monolithic step. This approach is motivated by the planner-executor paradigm, which has proven effective in various agentic reasoning tasks (Yao et al., 2023b; Erdogan et al., 2025; Biju et al., 2025). The core hypothesis is that by separating the high-level strategic planning from the low-level code implementation, the LLM can focus on each sub-task more effectively, reducing cognitive load and improving the quality of the final output.

This strategy was implemented as a distinct agent variant that overrides the standard AIDE operations (drafting, improving, and debugging) with a strict two-stage process.

Algorithmic Implementation

While the conceptual split is between a Planner and a Coder, our best agent implementation further decomposes the Coding Stage into a sequential, modular process. Inspired by CodeChain framework (Le et al., 2024). For any given task, the agent first invokes an LLM in a "Planner" role, followed by a second invocation in a "Coder" role.

Planning Stage: The Planner LLM receives the full context for the task (e.g., the problem description for a new draft, or the existing code and traceback for a debugging task). Its sole objective is to produce a detailed plan. For drafting, this is a detailed, step-by-step "Master Plan." For debugging, this is a "Bug Analysis" followed by a "Fix Plan." The Planner is explicitly forbidden from writing any implementation code. This stage is facilitated by a set of specialized prompts designed to elicit strategic thinking.

Coding Stage (Segmented Code Generation): The natural language plan generated by the Planner is then passed to the Coder LLM. The Coder's sole responsibility is to translate the provided plan into a complete, executable Python script. It is guided by its own set of prompts that instruct it to strictly adhere to the plan it was given (see Appendix B).

Our agent does not generate the entire script in one pass. Instead, after the Planner produces a single "Master Plan," the Coder is invoked iteratively to generate the solution segment by segment. The code is built up in a predefined, logical order (1. Setup & Imports, 2. Data Loading, 3. Preprocessing, etc.).

For each segment, the Coder is provided with the complete Master Plan and all previously generated code segments. Its task is to generate only the specific Python snippet relevant to the current segment. This new snippet is then appended to the script being built. This iterative process, transforms the single, complex task of writing a full program into a series of smaller, more manageable steps.

The final script produced by the Coder is then passed to the standard AIDE evaluation module. This entire multi-step process constitutes a single generative action (step) (a "draft" or "debug" step) within the AIDE solution tree. The workflow is outlined in Algorithm 2.

Algorithm 2 Decomposed Task Generation via Planner-Coder - The general purpose framework

Input: Initial Task Context T_C (e.g., problem description, buggy code)
Output: Final Solution Code C_{final}
▷ — Stage 1: Planning —
plan_prompt ← FormatPlannerPrompt(T_C)
plan ← LLM.GeneratePlan(plan_prompt) ▷ LLM returns a natural language plan
▷ — Stage 2: Coding —
code_prompt ← FormatCoderPrompt(T_C, plan)
C_{final} ← LLM.GenerateCodeFromPlan(code_prompt) ▷ LLM returns the full Python script
return C_{final}

3.5.3 Self-Consistency

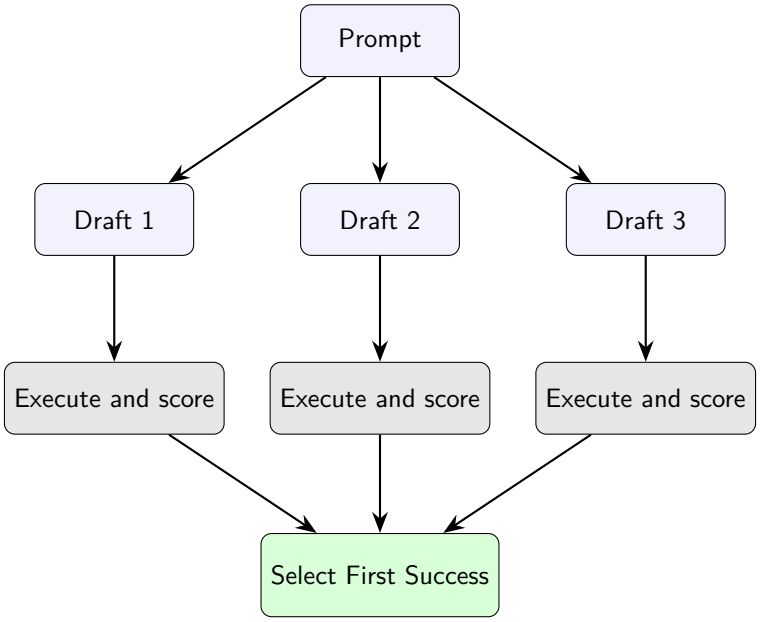


Figure 3.4: Self-Consistency: sample N diverse scripts, execute each, then pick the winner (first success or best metric).

The third ITS strategy implemented was Self-Consistency, designed to improve the robustness of the generation process by avoiding the impact of the LLM's stochasticity (Wang et al., 2022). While a single generative pass from a model can be unreliable, the core principle of Self-Consistency is that by sampling multiple, diverse outputs and aggregating them, the most common or highest-quality solution is more likely to be selected.

Our Self-Consistency Agent adapts this principle from its origins in other reasoning domains to the domain of executable program generation. The hypothesis is that by generating several distinct candidate scripts for a given task and then using the AIDE evaluation framework and execution feedback as a powerful verifier, we can filter out erroneous or suboptimal generations and increase the probability of committing a high-quality solution to the solution tree at each step.

Algorithmic Implementation

The Self-Consistency mechanism was integrated into all three primary generative operations of the AIDE agent: solution drafting, debugging, and improvement. To ensure a diverse set of outputs, a high sampling temperature (0.95) was used for the LLM queries within this agent. The workflow for any given generative step is detailed in Algorithm 3.

Candidate Generation: For a given task (drafting a new solution), the agent makes a single request to the Coder LLM, asking for N independent and diverse candidate solutions in parallel, where N is a configurable parameter (num_responses). This is achieved by setting the number of responses parameter in the API call to the vLLM backend. The backend then returns a list containing N candidate solutions.

Sequential Execution and Verification: The agent then iterates through this list of N candidates. Each candidate script is passed sequentially to the AIDE evaluation module, where it is executed and judged. The results (success/failure status and performance score) are stored.

Solution Selection: The final solution to be committed to the AIDE solution tree is chosen from the N verified candidates based on a pre-defined selection policy. Two policies were implemented to test different optimization goals:

First Success: This policy emulates a correctness-focused majority vote by selecting the first candidate in the sequence that executes without error. This prioritizes finding a working solution quickly.

Best Metric: This policy aligns with verifier-based approaches by evaluating all non-buggy candidates and selecting the one that achieves the best performance score, thus prioritizing solution quality.

Algorithm 3 Self-Consistency Agent Generative Step

Input: Initial Prompt P, Number of Candidates N, Selection Policy Π
Output: Selected Solution Node S_{best}
▷ Generate N diverse candidate solutions in a single, parallelized request
$CandidateList \leftarrow$ LLM.GenerateMultipleSolutions$(P, N,$ temperature$)$
$EvaluatedCandidates \leftarrow []$
for each $S_{candidate}$ in $CandidateList$ **do**
▷ Execute each candidate and have the feedback model parse the results
S_{eval}, score, is_buggy $\leftarrow$ ExecuteAndVerify$(S_{candidate})$
Add $(S_{eval},$ score, is_buggy$)$ to $EvaluatedCandidates$
▷ Select the best candidate based on the chosen verification policy
$S_{best} \leftarrow$ SelectBest$(EvaluatedCandidates, \Pi)$
return S_{best}

By generating and verifying multiple solution pathways using a solid feedback from the execution, the Self-Consistency agent increases the likelihood of AIDE producing a high-quality, non-buggy solution at each step, making the overall search process more robust.

3.6 Evaluation

To provide reliable evaluation of the inference-time strategies proposed in this thesis, a standardized evaluation protocol was established. This section outlines the benchmark competitions, the baselines for comparison, the specific metrics used for evaluation, and the overall experimental procedure. The primary goal is to establish a solid evaluation framework that can take a given agent configuration as input and produce clear, quantitative metrics to determine its effectiveness relative to other methods.

3.6.1 Benchmark Competitions

The foundation of our evaluation is a representative subset of competitions from the official MLE-Bench (Chan et al., 2024). While the full MLE-Bench is extensive, running experiments on all 75 competitions is computationally prohibitive for this project. Therefore, We selected a curated 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.

Representativeness: This subset is a good representation of the full 'lite' category and provides a strong, empirical basis for evaluation.

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 detailed in Table 3.2.

Table 3.2: Selected Benchmark Competitions from the 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

3.6.2 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 twofold: first, to 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	Proprietary / SOTA	Represents a strong, widely-used frontier model to establish a high-water mark for performance.
AIDE + o4-mini	Proprietary / Reasoning	A cost-effective, state-of-the-art reasoning model from OpenAI, serving as a key reference.
Our Model + Default AIDE	Open-Source /Reasoning	**Critical baseline.** This isolates the performance gain attributable specifically to our inference strategies by using our target model without any advanced ITS.
Human Performance	Historical Human Performance	Provides real-world context by comparing agent scores against the original Kaggle leaderboards.

3.6.3 Evaluation Metrics

To capture a holistic view of agent performance, we will use a number of metrics inspired directly by the MLE-Bench and AIDE papers as the standard for evaluating machine learning engineering agents. 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 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.

Experimental Procedure

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

Environment: All runs are executed within a Dockerized environment to ensure consistency. AIDE is configured with fixed hyperparameters for all methods (25 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.

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 three seeds was successful (a pass@k approach). The final percentages are averaged across the 10 competitions, with results reported as mean $\pm$ 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. Results and Analysis

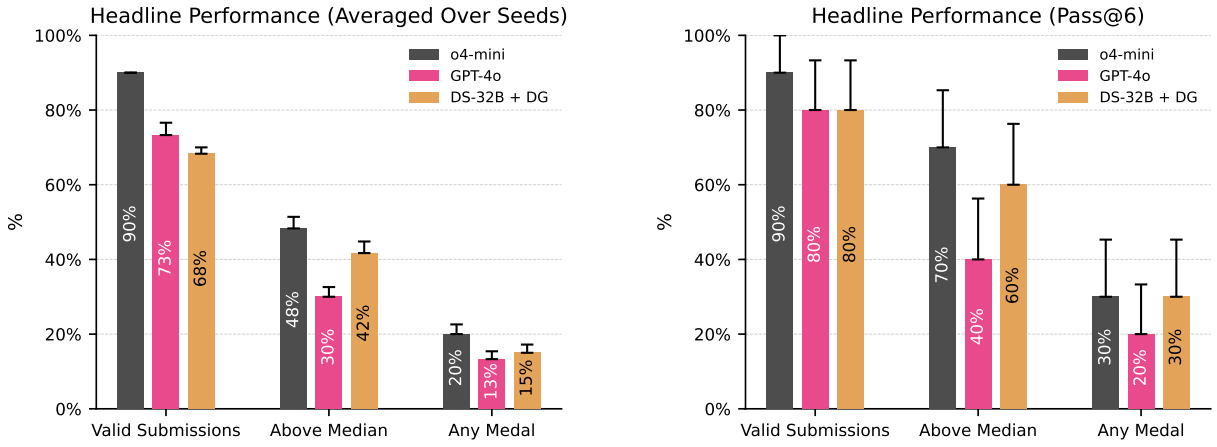
This chapter presents the empirical results of the experiments detailed in Chapter 3. The primary objective is to quantitatively assess the impact of the implemented Inference-Time Scaling (ITS) strategies on the performance of distilled DeepSeek models on a curated subset of the MLE-Bench benchmark.

We begin by presenting an aggregated overview of the performance of all evaluated agent configurations across our primary metrics. Subsequently, we analyze the performance of the baseline agent to establish a reference point, followed by a detailed, section-by-section analysis of the impact of each implemented ITS strategy: Self-Reflection, Decomposed Planner-Coder, and Self-Consistency.

Finally, we conclude with a comparative analysis that synthesizes these findings to identify the most effective strategies and key observation across different model scales.

4.1 Overview of Experimental Results

The complete experimental results are presented in this chapter, beginning with a high-level visual summary of the headline findings in Figure 4.1. We compare the per-attempt average performance against the Pass@6 success rate for our best-performing open-source agent (DeepSeek-32B with Decomposed Planner-Coder) and the proprietary baselines. Following this overview, we will delve into a detailed analysis of each ITS strategy.



(a) Average performance (mean ± SE).

(b) Headline performance (Pass@6).

Figure 4.1: Comparison of the top-performing agents. While the per-attempt performance (left) of the open-source agent (DS-32B + DG) lags behind the o4-mini baseline, its aggregated Pass@6 success rate (right) closes this gap, achieving an 'Any Medal' rate of 30%, which matches the o4-mini baseline.

To provide a more detailed breakdown, the aggregated performance of every model and agent configuration across the 10 selected MLE-Bench competitions is summarized in Table 4.1 and Table 4.2.

Table 4.1: Headline performance per attempt (seed–averaged). The best performing configuration for each model size is highlighted in **bold**.

Model	Strategy	Valid Submission (%)	Above Median (%)	Any Medal (%)
		Proprietary Baselines		
o4-mini	Baseline	**90.0 ± 0.0**	**48.3 ± 3.1**	**20.0 ± 2.6**
GPT–4-Turbo 2024-04-09	Baseline	73.3 ± 3.3	30.0 ± 2.6	13.3 ± 2.1
		DeepSeek 7B		
7B	Baseline	20.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
7B	Self-Reflection	21.7 ± 3.1	1.7 ± 1.7	0.0 ± 0.0
7B	**Self-Consistency**	**21.7 ± 4.8**	**1.7 ± 1.7**	**0.0 ± 0.0**
		DeepSeek 14B		
14B	Baseline	40.0 ± 3.7	5.0 ± 2.2	5.0 ± 2.2
14B	Self-Reflection	**48.3 ± 4.8**	11.7 ± 3.1	8.3 ± 3.1
14B	**Self-Consistency**	47.5 ± 6.3	**12.5 ± 4.8**	**10.0 ± 4.1**
		DeepSeek 32B		
32B	Baseline	63.3 ± 2.1	26.3 ± 3.1	11.7 ± 1.7
32B	Self-Reflection	55.0 ± 2.2	27.1 ± 2.9	5.0 ± 2.2
32B	Self-Consistency	**80.0 ± 0.0**	36.7 ± 3.3	10.0 ± 0.0
32B	**Decomposed Gen.**	68.3 ± 1.7	**41.7 ± 3.1**	**15.0 ± 2.2**
32B	PC + Self-Reflection	58.3 ± 3.1	23.3 ± 3.3	11.7 ± 1.7

Table 4.2: Pass@K performance of all evaluated ITS strategies across 10 MLE-Bench competitions. Where K is 6 attempts for competition. The best performing open-source configuration for each model size is highlighted in **bold**.

Model	Strategy	Valid Submission (%)	Above Median (%)	Any Medal (%)
Proprietary Baselines				
GPT-4o	Default AIDE	80.0	40.0	20.0
o4-mini	Default AIDE	**90.0**	**70**	**30.0**
DeepSeek 7B				
7B	Baseline	40.0	0.0	0.0
7B	Self-Reflection	50.0	10.0	0.0
7B	**Self-Consistency**	**50.0**	**10.0**	**0.0**
DeepSeek 14B				
14B	Baseline	60.0	10.0	10.0
14B	**Self-Reflection**	**80.0**	**30.0**	**20.0**
14B	Self-Consistency	70.0	30.0	20.0
DeepSeek 32B				
32B	Baseline	80.0	40.0	20.0
32B	Self-Reflection	60.0	10.0	
32B	Self-Consistency	80.0	10.0	
32B	PC + Self-Reflection		30.0	30.0
32B	**Decomposed Gen.**	**80.0**	**60.0**	**30.0**

4.2 Baseline Model Performance

Before evaluating the impact of our ITS strategies, it is essential to establish the baseline performance of the standard AIDE agent when powered by the distilled DeepSeek models. These results, presented in Table 4.1 and Table 4.2, quantify the initial capabilities of the models and highlight the performance gap that our research aims to address.

A clear and predictable trend emerges related to model scale. As shown in Table 4.2 (Pass@6), the 7B parameter model is functionally unable to solve any of the benchmark tasks to a medal-winning standard, achieving zero medals "Any Medal". Its core limitation lies in its fundamental code generation capability; with a per-attempt valid submission rate of only 20.0% (Table 4.1), the agent fails to produce an evaluatable solution in four out of five attempts. Qualitative analysis of the run logs for the 7B model reveals that these failures are most frequently due to basic coding errors, such as FileNotFoundError from incorrect path specifications, or SyntaxError and AttributeError from the misuse of common library APIs. This suggests that while the model may possess reasoning capabilities from its distillation, it lacks the specialized coding proficiency to reliably translate plans into executable scripts.

Performance scales consistently with model size. The 14B model represents a significant step up, achieving a 10.0 "Any Medal" rate (Pass@6) and more than doubling the per-attempt valid submission rate of the 7B model to 40.0%.

The 32B model is by far the most capable of open-source baselines, achieving an "Any Medal" rate of 20.0 (Pass@6) and a per-attempt valid submission rate of 63.3%. This level of performance is comparable to the baseline GPT-4-Turbo, although the GPT-4-turbo is slightly better than the DeepSeek-32B on average. However, a notable performance gap remains compared to one of the state-of-the-art reasoning models, o4-mini, which achieves a 30.0% "Any Medal" rate. This 10-percentage point gap quantifies the central challenge for this thesis: to determine whether inference-time scaling strategies can bridge this gap and elevate the performance of the accessible 32B model to be comparable to models of GPT-4-turbo and o4-mini sizes.

The key takeaway from the baselines experiments is that, for DeepSeek reasoning models, the size of the model is a significant factor in the overall behavior of the model. another observation on the performance of the DeepSeek 7B model is that it was originally trained specifically for math, i.e, it is not well equipped to perform coding tasks unless fundamentally fine-tuned, which makes employing ITS at this low level a difficult task, or at least, limited potential. On the other side, very good signals are present from the DeepSeek-32B, This is largely due to the fact that this model has the ability to handle longer context, follow the output structure specified "this is very critical as all of agentic loops depends on the model's adherence to output format", in out case, we used regular expressions to extract code, and DeepSeek-7B does not comply with the output format, and frequently produces malformed outputs.

4.3 Quantitative and Qualitative Impact of Self-Reflection

4.3.1 Quantitative Impact

The impact of Self-Reflection (SR) is most clearly illustrated by comparing the performance of SR-enabled agents to their baselines in Table 4.1. For the 7B model, the effect was negligible. The per-attempt Valid Submission Rate increased only marginally from 20.0% to 21.7%, and this minor improvement in reliability did not translate into any competitive solutions, as the **Any Medal** rate remained at 0.0%.

In contrast, the 14B model represents the "sweet spot" for this strategy. The application of SR provided a clear and significant performance boost. The per-attempt Valid Submission Rate rose from 40.0% to 48.3%, and more importantly, the *Any Medal* rate nearly doubled from 5.0% to 8.3%. This trend is confirmed in the Pass@6 results (Table 4.2), where the medal rate for the 14B model also doubled from 10.0% to 20.0%, demonstrating a substantial improvement in its overall problem-solving capability.

However, for the largest 32B model, the SR strategy appeared to be detrimental to headline performance. While the *Above Median* rate saw a slight uptick, the per-attempt **Any Medal** rate was more than halved, dropping from 11.7% for the baseline to just 5.0% for the SR agent.

4.3.2 Qualitative Analysis

A qualitative analysis of the agent execution logs reveals the narrative behind these divergent results. The failure of SR on the 7B model can be attributed to the model's fundamental weakness in adhering to strict output formatting. In a representative run, the agent repeatedly failed to generate a clean, executable script. The SR mechanism correctly identifies **SyntaxError** at each step, but the subsequent revisions were unable to correct this foundational flaw, resulting in a high-cost loop of repeated failure.

The success of SR on the 14B model suggests that it is most effective when the base model is capable of producing a reasonable first draft but still makes rectifiable errors. In these cases, the SR loop successfully guides the agent to fix specific, contained bugs like API misuse or minor logical flaws, which directly translates to the observed increase in both reliability and solution quality.

The negative impact of SR on the 32B model highlights a critical insight: for an already capable model, an untargeted reflection step can introduce "overthinking," where the agent modifies correct code and inadvertently introduces regressions.

Ultimately, these findings indicate that Self-Reflection is not a universally beneficial strategy. Its success is contingent on the base model possessing a sufficient level of foundational competence without being so advanced that the reflection process becomes counterproductive. It is a powerful tool for correcting specific flaws but can be a counterproductive if not carefully applied.

4.4 Impact of Self-Consistency (SC)

4.4.1 Quantitative Impact

Self-Consistency (SC) trades additional inference cost for reliability by generating multiple candidate solutions and selecting the first that passes an internal verification check.[1] Table 4.1 captures the per-attempt headline results, while Table 4.2 shows the more forgiving Pass@6 view.

7B model. For the smallest distilled model, SC had no meaningful effect. The Valid Submission Rate remained flat at 21.7 %, and the **Any Medal** rate stayed at 0 %. Because each individual draft already fails on basic syntax or API usage, the "generate three" strategy simply multiplies failures without producing a valid solution.

14B model. SC produces a clear uplift over both the baseline and SR variants. The Valid Submission Rate (47.5 %) is roughly on par with SR, but the **Above Median** rate rises to 12.5 % and—more importantly—the **Any Medal** rate improves to 10.0 % (double the baseline's 5.0 %). The Pass@6 table reinforces this: medals climb from 10.0 % to 20.0 %. These gains suggest that the 14B model often

[1] In our implementation $k{=}3$ parallel drafts are produced; the first draft whose unit tests pass is submitted.

generates at least one correct draft within three attempts if given the opportunity to explore small variations.

32B model. SC yields the highest Valid Submission Rate of all open-source configurations (80.0 %), surpassing both the baseline (63.3 %) and SR (55.0 %). Surprisingly, this does *not* translate into more medals—**Any Medal** drops slightly to 10.0 %, half the baseline's 20.0 %. The pattern repeats in the Pass@6 view. The data imply that SC helps this large model avoid outright failure but that the "first-success" policy often locks in a merely acceptable draft rather than the best one. the cost of changing the selection strategy for this model will increase the time for inference to the double, so the tradeoff can be made in order to get better results, but our experiments sought to standardize the policy across the three models.

4.4.2 Qualitative Analysis

Log inspections clarify why SC works at 14B yet disappoints at 32B:

- **7B.** All three drafts commonly share the same structural errors (e.g., malformed CSV paths, incorrect Pandas imports). With no working draft to choose from, SC cannot rescue the run.
- **14B.** This model frequently produces one near-correct script alongside two defective ones. SC's verification gate filters out the bad drafts and selects the workable candidate, explaining the doubled medal rate.
- **32B trade-off.** The largest model routinely generates *several* scripts that pass unit tests but differ in optimisation choices (e.g. choice of CatBoost vs. LightGBM). Because we submit the *first* passing draft, SC may lock in a quick-but-suboptimal solution, losing leaderboard gains the baseline sometimes achieves after a more extensive single-draft search.

Qualitativly, the key take-awayis that SC is most helpful when the model:

1. can already produce at least one nearly correct draft (14B), and
2. benefits from multiple *diverse* attempts because errors are local, not systemic.

For very weak coders (7B), no amount of parallel drafting overcomes fundamental gaps. For very strong coders (32B), simply accepting the first pass may cap performance, nevertheless, compared to basline performance, the self-consistency ITS strategies bring huge gains and notable improvements. Future work might explore *rank-then-select* policies—e.g. evaluate all k drafts on the validation subset and select the best—to realise SC's reliability gains without sacrificing leaderboard potential.

4.5 Impact of Decomposed Task Generation

To address the common failure mode of logical incoherence in monolithically generated scripts, we evaluated a Decomposed Planner-Coder strategy. This approach enforces a planner-coder paradigm, where a high-level plan is generated first, followed by a separate code generation stage. Due to computational constraints, this strategy was benchmarked exclusively on the 32B DeepSeek model.

4.5.1 Quantitative Impact

The Decomposed Task Generation strategy yielded the best overall performance of any open-source configuration tested. As shown in Table 4.2, this agent achieved an Any Medal rate of 30.0% (Pass@6),

a significant 10 percentage point improvement over the 32B baseline and a 20 percentage point improvement over the 32B Self-Reflection agent. This result elevates the 32B model to be directly competitive with the o4-mini proprietary baseline, demonstrating that a superior agentic workflow can close the performance gap.

However, attempts to augment this already strong strategy proved counterproductive. The hybrid PC + Self-Reflection agent, which added a reflection step after code generation, saw its Any Medal rate fall sharply from 30.0% back down to 10.0

4.5.2 Qualitative Analysis

Log analysis reveals that the success of this strategy stems from its ability to enforce architectural coherence. In monolithic generation, models would often produce plans that mixed incompatible libraries (e.g., planning to use scikit-learn but then generating PyTorch training code). By separating the planning and coding stages, the agent is forced to first commit to a consistent, high-level blueprint.

The failures observed with this agent typically occurred when the initial plan was itself flawed or ambiguous. However, when the Planner produced a clean, single-paradigm plan (e.g., "use scikit-learn exclusively"), the Coder was able to faithfully implement it, resulting in a correct and high-performing script. This decomposition provides a robust framework that allows the agent to overcome common failure modes like library confusion and converge on a logically consistent solution. A detailed case study of this process is provided in Appendix A.

4.6 Comparative Analysis and Key Findings

A comparative analysis of the experimental results reveals several key findings regarding the efficacy of different ITS strategies and their interaction with model scale. The headline performance metric, Any Medal % (Pass@6), is visualized in Figure 4.2 for the best-performing configuration of each open-source model against the proprietary baselines.

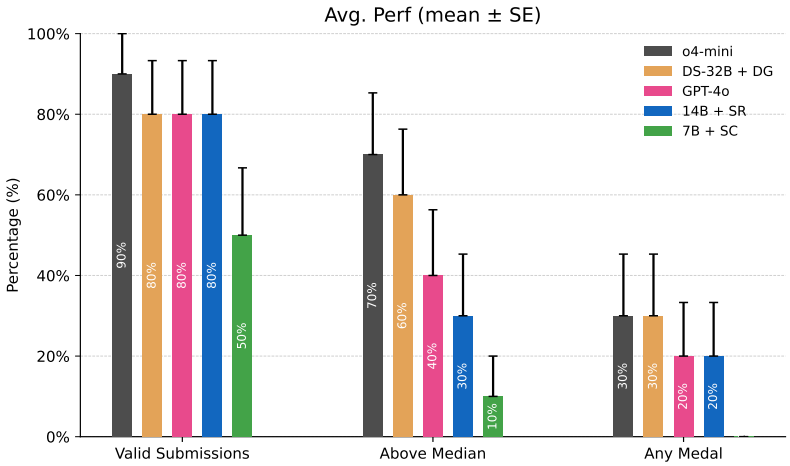


Figure 4.2: Comparison of the headline "Any Medal % (Pass@6)" metric across the best-performing agent configurations. The Decomposed Planner-Coder strategy with the 32B model successfully closes the gap to the proprietary o4-mini baseline.

This synthesis of our quantitative and qualitative results leads to four primary conclusions:

- **ITS Cannot Rescue Foundational Model Deficiencies**. The effectiveness of the evaluated ITS strategies is fundamentally restricted by the core capabilities of the base model. For the 7B model, which was originally optimized for mathematical reasoning, its inherent limitations in code generation for complex ML engineering tasks proved to be a big issue. Although strategies like Self-Consistency could slightly increase the valid submission rate, they could not generate a single-medal-winning solution. This suggests that when a model's foundational deficits are too great, ITS strategies may fail to provide meaningful uplift on complex, multi-step tasks like ML engineering, even if they might be effective on simpler, single-pass generation tasks.
- **Task Decomposition is a Highly Effective, but Costly, Strategy for Capable Models**. For the 32B model, the Decomposed Task Generation (Planner-Coder) strategy was the clear top agent-ITS configuration. This approach scored an Any Medal rate of 30.0% (Pass@6), significantly outperforming the 32B baseline (20.0%) and surpassing the powerful GPT-4-Turbo baseline (20.0%). This result is on par with the state-of-the-art o4-mini, demonstrating that a sophisticated agentic workflow can elevate an accessible open-source model to a world-class performance level. However, this performance comes at a computational cost, as the number of LLM calls scales linearly with the number of segments in the generated code.
- **There is no "one-size-fits-all" ITS strategy**. Our results confirm findings in the broader literature: the optimal ITS strategy is highly dependent on both the model and the task. Self-Reflection, for example, was highly effective for the "competent but flawed" 14B model, but was detrimental to the more capable 32B model. Similarly, Self-Consistency provided a crucial reliability boost for the 14B model but acted as a performance cap on the 32B model due to its "first-success" policy that we used for this method. This highlights the need for adaptive agentic systems. Although our framework supports switching between strategies via a configuration flag, the dynamic selection of the best ITS strategy based on the task context remains a key challenge for future work.
- **Agentic frameworks are highly sensitive to prompt quality.** A crucial, overarching finding from this research is the profound impact of prompt engineering and context management. A significant portion of the project effort was devoted to crafting precise, unambiguous prompts for each agent role and ITS strategy (see Appendix B). The success of the "Decomposed Task Generation" strategy, for instance, depended heavily on the clarity and completeness of the plan produced in its first stage. As agentic systems grow more complex, the quality of the natural-language instructions that guide them emerges as a critical determinant of performance.

5. Discussion and Conclusion

This research set out to investigate whether Inference-Time Scaling (ITS) strategies, proven effective in mathematical reasoning, could elevate the performance of accessible, open-source language models on complex, end-to-end machine learning engineering tasks. By implementing and evaluating a suite of ITS agents on the DeepSeek-R1 distilled models within the AIDE framework, we have clear evidence to answer our primary research questions. This final chapter discusses the implications of our findings, acknowledges the limitations of this study, proposes directions for future research, and offers a concluding summary.

5.1 Discussion of Key Findings

Our experimental results, detailed in Chapter 4, lead to an understanding of the interplay between model scale, agentic strategy, and task complexity. We can now directly address the research questions posed in Chapter 1.

5.1.1 Answering the Research Questions

1. The Efficacy of ITS on Distilled Models for ML Engineering. Our first research question asked to what extent ITS strategies can improve the performance of distilled DeepSeek models on ML engineering tasks. Our findings show that the impact is significant but highly dependent on the base model's foundational capabilities. For the 7B model, which exhibited fundamental flaws in code generation, ITS strategies offered only marginal gains in reliability and failed to produce any competitive solutions. However, for the 14B and 32B models, the impact was profound. The best-performing strategies doubled the Any Medal rate of the 14B model (from 10.0% to 20.0% Pass@6) and elevated the 32B model to be competitive with state-of-the-art proprietary models. This confirms that ITS is a powerful tool, but it acts as a performance amplifier, and not for a model that is unsuited for the core task.

2. Performance Trade-offs Between ITS Strategies. Our second research question concerned the trade-offs between different ITS strategies. We found no single "best" strategy; instead, each demonstrated a unique performance profile.

Self-Reflection (SR) proved most effective for the mid-sized 14B model, where it excelled at fixing the contained, rectifiable errors common to that model scale. However, it was detrimental to the 32B model, where it often led to "overthinking" and the regression of correct code.

Self-Consistency (SC) emerged as a powerful strategy for improving the reliability and Valid Submission Rate of all models. However, its "first-success" selection policy created a trade-off between reliability and quality, as it sometimes locked in suboptimal solutions for the highly capable 32B model.

Decomposed Task Generation (Planner-Coder) was the clear top-ITS for the 32B model. Its primary benefit was in enforcing architectural coherence and leaving small room for error by generating using segments and detailed planning, which proved crucial for solving complex tasks. Its cost, however, is a linear increase in LLM calls with the number of generated code segments.

3. Comparison to Proprietary Baselines. Our third research question asked how the enhanced open-source models compare to proprietary baselines. Our results demonstrate that with the right scaffolding,

the performance gap can be closed. The baseline DeepSeek-32B model was competitive with GPT-4-Turbo but lagged significantly behind o4-mini. However, when augmented with the Decomposed Task Generation strategy, the DeepSeek-32B agent achieved an Any Medal rate of 30.0% (Pass@6), matching the performance of the o4-mini baseline. This is the most significant finding of this work, proving that a sophisticated agentic workflow can make an accessible, open-source model competitive at the highest level.

5.2 Limitations of the Study

While this research provides valuable insights, it is important to acknowledge its limitations:

Limited Benchmark Scope: The experiments were conducted on a curated subset of 10 "lite" complexity competitions from MLE-Bench. The findings may not generalize to the more complex "medium" and "high" difficulty tasks, which require handling larger datasets and more intricate modeling. Nevertheless these finding show clear improvement, suggesting a potential for extrapolation if benchmarked over a lrger set of competitions **Quantization Effects**: To make the experiments feasible on our hardware, 8-bit (Floating Point 8-FP8) quantized versions of the DeepSeek models were used. It is possible that the use of full-precision models could alter the performance and behavioral characteristics of the agents.

Unevaluated Strategies: Due to computational constraints, several implemented strategies, most notably a Tree of Thoughts (ToT) agent, were not fully benchmarked. The ToT agent, designed to explore multiple high-level plans in parallel, represents a promising but costly approach that was beyond the scope of this project. we also did not manage to benchmark the decomposed Task Generation on the other two models for the same reasons.

5.3 Future Work

The findings and limitations of this study suggest several promising directions for future research:

Dynamic ITS Strategy Selection: A key finding was that no single ITS strategy was optimal for all models or situations. Future work should focus on creating a meta-agent capable of dynamically selecting the most appropriate strategy (e.g., use Self-Consistency for initial exploration, switch to Self-Reflection for debugging, and use a Planner-Coder for complex improvements) based on the current state of the solution tree.

Evaluation of Advanced Agents: A full-scale evaluation of the implemented but un-benchmarked Tree of Thoughts (ToT) agent is a critical next step, in addition to implementing more ITS techniques to complete the study comprehensively. This would provide insights into whether the high computational cost of a planning-stage search translates to superior performance on the most complex tasks.

Richer Feedback Mechanisms: The current AIDE framework relies on a feedback model to interpret execution tracebacks. Future work could explore integrating more direct feedback, such as using static analysis tools to identify potential bugs before execution or leveraging a debugger to provide more granular, line-by-line feedback to the agent.

5.4 Conclusion

This thesis presented a systematic evaluation of Inference-Time Scaling strategies on reasoning-distilled, open-source language models for the complex domain of ML engineering. We have demonstrated that while smaller models have inherent limitations that ITS cannot fully overcome, a well-structured agentic workflow can dramatically amplify the capabilities of larger, more competent open-source models.

Our key contribution is the finding that a 32B parameter DeepSeek model, when augmented with a Decomposed Code Generation strategy, achieves performance on par with a state-of-the-art proprietary model on the rigorous MLE-Bench benchmark. This confirms our central hypothesis that performance lies not just in building larger models, but in creating more intelligent and sophisticated agentic scaffolding around the powerful open-source models that are already available today.

Acknowledgements

First and foremost, I express my deepest gratitude to the African Institute for Mathematical Sciences (AIMS) for providing the environment, resources, and support necessary for this research. Special thanks to my supervisors, Arnol Fokam and Arnu Pretorius, for their invaluable mentorship, insightful guidance, and continuous support throughout the project.

My heartfelt thanks also go to my family and friends, whose constant encouragement and unwavering belief have sustained me during this academic journey.

Finally, I dedicate this work to my beloved country, with earnest hopes and prayers for peace, unity, and freedom from conflict. May the spirit of resilience and determination lead us to brighter, peaceful days ahead

Bibliography

AI, R. H. Deepseek-r1 distill qwen 32b fp8 dynamic — model card. https://huggingface.co/RedHatAI/DeepSeek-R1-Distill-Qwen-32B-FP8-dynamic, 2025. Accessed 2025-06-10.

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

Anthropic. Claude: A constitutional ai assistant. https://www.anthropic.com/index/2023/claude-constitutional-ai, 2023. Accessed: 2025-06-09.

Anthropic. Claude 3.5 sonnet, 2024. URL https://claude.ai/. Retrieved June 13, 2025 from conversation with Claude 3.5 Sonnet.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., and Dohan, D. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021a.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Rosca, M., Krueger, D., and Sutton, C. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021b.

Bai, Y., Kadavath, S., Kundu, S., Askell, A., Kernion, J., Jones, A., Chen, A., Goldie, A., Mirhoseini, A., McKinnon, C., et al. Constitutional ai: Harmlessness from ai feedback. *arXiv preprint arXiv:2212.08073*, 2022.

Bai, Y., Zhang, Y., Yang, Z., Shao, Z., Liu, K., Wang, R., Wang, Y., Yan, Z., Huang, Z., Li, H., et al. Qwen: A family of high-performance open-source language models. *arXiv preprint arXiv:2312.14072*, 2023. URL https://arxiv.org/abs/2312.14072.

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

Biju, E., Talaei, S., Huang, Z., Pourreza, M., Mirhoseini, A., and Saberi, A. Sprint: Enabling interleaved planning and parallelized execution in reasoning models, 2025. URL https://arxiv.org/abs/2506.05745.

Bommasani, R., Hudson, J., Adeli, E., Altman, R., Arora, S., von Arx, A., Bernstein, M., Bohg, J., Bosselut, A., Brunskill, E., et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

Brown, T., Chen, Z., and Mann, B. Revisiting few-shot learning with test-time compute. *Proceedings of the 41st ICML*, 2024.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33:1877–1901, 2020.

Chan, S., Chowdhury, N., Jaffe, O., Aung, J., Sherburn, D., Mays, E., Starace, G., Liu, K., Maksin, L., Patwardhan, T., Weng, L., and Madry, A. Mle-bench: Evaluating machine learning agents on machine learning engineering. *arXiv preprint arXiv:2410.07095*, 2024. URL https://arxiv.org/abs/2410.07095.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Chen, W., Ma, X., Wang, X., and Cohen, W. W. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*, 2022.

Chen, X., Lin, M., Schärli, N., and Zhou, D. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*, 2023.

Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., Hesse, C., and Schulman, J. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021. URL https://arxiv.org/abs/2110.14168.

Cognition. Introducing devin, the first ai software engineer. Blog Post, March 2024. URL https://www.cognition-labs.com/blog/introducing-devin.

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

DeepMind, T. Inference-time budget allocation beats parameter scaling. *Nature Machine Intelligence*, 2025.

Du, Y., Li, S., Torralba, A., Tenenbaum, J. B., and Mordatch, I. Improving factuality and reasoning in language models through multiagent debate. *arXiv preprint arXiv:2305.14325*, 2023.

Ehrlich, J., Nijkamp, E., and Berent, I. Self-consistency with reflection improves math solving. In *ACL*, 2025.

Erdogan, L. E., Lee, N., Kim, S., Moon, S., Furuta, H., Anumanchipalli, G., Keutzer, K., and Gholami, A. Plan-and-act: Improving planning of agents for long-horizon tasks. *arXiv preprint arXiv:2503.09572*, 2025.

Guo, D., Yang, D., Zhang, H., Song, J., Zhang, R., Xu, R., Zhu, Q., Ma, S., Wang, P., Bi, X., Zhang, X., Yu, X., Wu, Y., Wu, Z. F., Gou, Z., Shao, Z., Li, Z., Gao, Z., Liu, A., Xue, et al. Deepseek-R1: Incentivizing reasoning capability in llms via reinforcement learning. arXiv preprint arXiv:2501.12948, 2025a. URL https://arxiv.org/abs/2501.12948.

Guo, Y., Zhang, W., and Liu, J. Scaling test-time compute for reliable mathematical reasoning. *Transactions on Machine Learning Research*, 2025b.

Hendrycks, D., Burns, C., Kadavath, S., Arora, A., Basart, S., Tang, E., Song, D., and Steinhardt, J. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*, 2021. URL https://arxiv.org/abs/2103.03874.

Hoffmann, J., Borgeaud, S., Mensch, A., Buchatskaya, E., Cassirer, T., Gabriel, S., Rutherford, S., Millican, K., Driessche, G. v. d., Lespiau, J., et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022. URL https://arxiv.org/abs/2203.15556.

Jemison, C. E., He, A. Q., Cassano, F., Sengupta, S., Yu, M., and Pasupat, P. Swe-bench: Can language models solve real-world software engineering problems? *arXiv preprint arXiv:2401.10744*, 2024.

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

Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., and Amodei, D. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, 2023.

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

Le, H., Chen, H., Saha, A., Gokul, A., Sahoo, D., and Joty, S. Codechain: Towards modular code generation through chain of self-revisions with representative sub-modules, 2024. URL https://arxiv.org/abs/2310.08992.

Li, X. and Wu, L. Reward-model pruning for efficient proof search. In *ICLR*, 2024.

Li, Y., Choi, D., Chung, J., Kushman, N., by step, S., et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022a.

Li, Y., Choi, J., FitzGerald, J., Freeman, D., Gkatzia, D., Irving, G., Li, Y., Mann, T., Menick, J., Ring, M., et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022b. doi: 10.1126/science.abq1158.

Liu, P., Chen, H., and Xin, J. Beyond string matching: Evaluating code generation via execution. *Empirical Methods in Natural Language Processing (EMNLP)*, 2023.

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.

Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., et al. Self-refine: Iterative refinement with self-feedback. *arXiv preprint arXiv:2303.17651*, 2023.

Muennighoff, N., Bajaj, P., et al. Parallel sampling and verifier search for small-scale llms. *arXiv preprint arXiv:2504.01234*, 2025.

NVIDIA Corporation. Nvidia hopper architecture in-depth. https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/, 2023. Accessed 2025-06-10.

NVIDIA Corporation. Tensorrt-llm supercharges large language model inference on h100 gpus. https://developer.nvidia.com/blog/nvidia-tensorrt-llm-supercharges-large-language-model-inference-on-nvidia-h100-gpus/, 2024. Accessed 2025-06-10.

OpenAI. Humaneval: Hand-written evaluation set for evaluating code generation models. https://github.com/openai/human-eval, 2021. Dataset, accessed 13 June 2025.

OpenAI. Optimizing inference compute for gpt models. https://openai.com/research/test-time-scaling, 2024. Accessed 2025-06-09.

Qwen, T. Qwenlm technical report. https://github.com/QwenLM, 2024. Accessed 2025-06-09.

Renze, M. and Guven, E. The benefits of a concise chain of thought on problem-solving in large language models. In *2024 2nd International Conference on Foundation and Large Language Models (FLLM)*, page 476–483. IEEE, Nov. 2024. doi: 10.1109/fllm63129.2024.10852493. URL http://dx.doi.org/10.1109/FLLM63129.2024.10852493.

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.

Shinn, N., Labash, B., and Gopinath, A. Reflexion: an autonomous agent with dynamic memory and self-reflection. *arXiv preprint arXiv:2303.11366*, 2023.

Snell, C. V., Lee, J., Xu, K., and Kumar, A. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024a.

Snell, J., Deng, B., and Paxton, C. Verifier-guided tree search for llm reasoning. *NeurIPS*, 2024b.

Sun, Y., Li, Y., Dong, Y., Alon, U., Lee, W., Neubig, G., and Hellendoorn, V. J. S*: A framework for augmenting parallel sampling with sequential debugging. *arXiv preprint arXiv:2402.04696*, 2024.

Tian, Y., Peng, B., Song, L., Jin, L., Yu, D., Mi, H., and Yu, D. Toward self-improvement of llms via imagination, searching, and criticizing. *arXiv preprint arXiv:2408.03314*, 2024. URL https://arxiv.org/abs/2408.03314.

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

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, , and Polosukhin, I. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://papers.nips.cc/paper_files/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

Wang, X., Wei, J., Schuurmans, D., Le, Q. V., Chi, E. H., Narang, S., Chowdhery, A., and Zhou, D. Self-consistency improves chain-of-thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022. URL https://arxiv.org/abs/2203.11171. Published at ICLR 2023.

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

Wei, J., Tay, Y., Bommasani, R., Raffel, C., Zoph, B., Borgeaud, S., Yogatama, D., Fedus, W., chowdhery, a., et al. Emergent abilities of large language models. *Transactions on Machine Learning Research*, 2022.

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

Wu, Y., Wang, Z., Zhang, X., Chen, K., Yang, L., Tan, M., Liu, C., Zhang, Y., Dai, X., and Yuan, L. A survey on efficient inference for large language models. *arXiv preprint arXiv:2404.14294*, 2024. URL https://arxiv.org/abs/2404.14294.

Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T., Cao, Y., and Narasimhan, K. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822, 2023a.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. React: Synergizing reasoning and acting in language models, 2023b. URL https://arxiv.org/abs/2210.03629.

Zaharia, M., Khattab, O., Chen, L., Davis, J. Q., Miller, H., Potts, C., Zou, J., Carbin, M., Frankle, J., Rao, N., and Ghodsi, A. The shift from models to compound ai systems. *Blog: BAIR*, 2024. URL https://bair.berkeley.edu/blog/2024/02/18/compound-ai-systems/.

Zhao, W. X., Zhou, K., Li, J., Tang, T., Wang, X., Hou, Y., Min, Y.-c., Zhang, B., Zhang, J.-j., Dong, Z., et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023.

Appendix A. Qualitative Case Studies of Agent Behavior

This appendix provides detailed, qualitative analyses of specific experimental runs to illustrate the agent behaviors discussed in Chapter 4. By examining the verbose logs and `journal.json` files, we can move beyond aggregate statistics to understand the narrative of each problem-solving attempt. These case studies offer concrete evidence for the quantitative findings, highlighting the specific failure modes and success patterns of each ITS strategy.

A.1 Case Study: Foundational Limits of the 7B Model

This case study examines two runs of the DeepSeek-7B model on the "Random Acts of Pizza" competition: the baseline agent and the Self-Reflection (SR) agent. The logs reveal that, for a model with fundamental deficiencies in code generation and instruction follow-up, even an advanced ITS strategy like SR can be ineffective.

A.1.1 The Baseline Agent: A Narrative of Environmental and Conceptual Failure

The baseline run was characterized by an inability to overcome the most basic setup tasks. It failed before it could execute any meaningful machine learning logic, demonstrating a critical lack of environmental awareness and task comprehension.

Listing 1 The 7B baseline agent's trajectory, which failed due to a combination of unresponsive model queries, a complete misinterpretation of the task, and basic file system errors.

```
# LOG: Initial Failures of the Baseline Agent
# Attempt 1: Model backend is unresponsive
# TRACEBACK:
# Request timed out

# Attempt 2: Conceptual Failure
# AGENT'S PLAN:
# Hypothesizes about extracting features from image filenames,
# despite this being a text classification task.

# Attempt 3: Execution Failure
# AGENT'S CODE:
df = pd.read_csv("train.csv") # Incorrect path
# TRACEBACK:
# FileNotFoundError: [Errno 2] No such file or directory: 'train.csv'
```

A.1.2 The Self-Reflection Agent: A Narrative of Repetitive Failure

In contrast, the SR agent correctly identified the high-level goal and formulated a reasonable plan. However, its journey was a consistent cycle of generating syntactically invalid code and being unable to correct it, even with the reflection mechanism.

The Core Error: Persistent Format Violation. The agent's primary failure was its inability to adhere to the required output format. It repeatedly mixed its narrative thinking directly into the executable code block, causing an immediate `SyntaxError`.

Listing 2 The 7B-SR agent's initial attempt, which fails due to a basic syntax error caused by including narrative text in the code block.

```
# LOG: Initial buggy code generation in Step 1
<execute_ipython>
# Plan:
# 1. Load the training and test data.
# 2. Preprocess the text data using TF-IDF.
# ...
import pandas as pd
train = pd.read_json('input/train.json')
# ...
<end_execute_ipython>

# TRACEBACK from interpreter:
#   File "<stdin>", line 2
#     # Plan:
#     ^
# SyntaxError: invalid syntax
```

The Ineffective Reflection Loop. The SR mechanism was correctly triggered by this error, and the "Reviewer" role accurately identified the bug. However, the "Coder" role, when tasked with the revision, repeated the exact same formatting mistake, demonstrating a fundamental flaw in its generative process.

Listing 3 The SR agent identifies the `SyntaxError` but repeats the same mistake in its revision, showing that the reflection mechanism could not fix the underlying format violation.

```
# LOG: SR critique after Step 1
The code provided contains a syntax error because it includes the plan as comments
within the executable code block. The Python interpreter does not allow this.
I need to remove the plan and only provide the code.

# LOG: Revised (and still buggy) code generated in the next step
# The agent generates the exact same malformed output, repeating the error.
<execute_ipython>
# Plan: ...
# ...
<end_execute_ipython>
```

Revealing Deeper Flaws. As this cycle of failure continued, the agent's attempts to vary its implementation revealed deeper weaknesses in its understanding of common data science libraries.

- **API Misuse:** The agent attempted to use `OneHotEncoder` on text features that were lists of strings, resulting in a `TypeError: unhashable type: 'list'`. This shows a shallow understanding of the library's constraints.

- **Hallucination:** In a later step, the agent confidently generated code to import a non-existent function, `from sklearn.preprocessing import SparseStandardScaler`, a classic LLM hallucination.

A.1.3 Conclusion of Case Study

This comparison highlights that for a model with foundational weaknesses like the DeepSeek-7B variant, an ITS strategy like Self-Reflection is insufficient. The baseline agent failed on basic environmental awareness. The SR agent failed due to a more complex but equally fundamental issue: an inability to follow core formatting instructions, which its own reflection mechanism could diagnose but not correct. This suggests that for certain models, the generative priors are so strong that they override explicit instructions, and no amount of self-correction can fix the problem.

A.2 Case Study: Impact of Self-Reflection on DeepSeek-14B

In contrast to the 7B model, the DeepSeek-14B model demonstrates a significantly higher level of foundational capability. This case study, comparing the baseline and SR agents on the "Spooky Author Identification" task, reveals how Self-Reflection can act as a powerful debugger for a model that is "competent but flawed."

A.2.1 The Baseline Agent: A Catastrophic Conceptual Failure

The baseline 14B agent's run was a non-starter due to a complete misinterpretation of the task. Despite receiving the correct prompt, the agent hallucinated an entirely different problem, demonstrating profound context-blindness.

Listing 4 The 14B baseline agent completely misunderstood the task, hallucinating a different dataset and failing immediately. It never recovered.

```
# LOG: 14B-Baseline agent's first execution attempt
# AGENT'S PLAN: (The agent's internal monologue focuses entirely on wine quality)
# "The task is to predict wine quality... I will use RandomForestClassifier..."

# AGENT'S CODE:
import pandas as pd
df = pd.read_csv("wine_dataset.csv") # <-- This file does not exist.

# TRACEBACK:
# FileNotFoundError: [Errno 2] No such file or directory: 'wine_dataset.csv'
```

A.2.2 The Self-Reflection Agent: A Narrative of Successful Debugging

The SR-enabled agent, by contrast, correctly understood the task from the beginning. While its path was not free of errors, it successfully used the critique-and-revise loop to debug its own code and converge on a high-quality solution. This journey illustrates the value of SR as an effective debugging mechanism.

Example: Overcoming API Errors. In an early step of its run, the agent generated code with a subtle but fatal `AttributeError`, using a plausible but incorrect method name.

Listing 5 An initial buggy attempt by the 14B-SR agent.

```
# LOG: ds-14b-sr agent's buggy code in Step 4
# ... text preprocessing ...
punc = '!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
# The agent hallucinates "mtable" instead of the correct "maketrans"
transtab = str.mtable(punc, ' '*len(punc))
# ...
# TRACEBACK:
# AttributeError: type object 'str' has no attribute 'mtable'
```

The SR mechanism was able to catch this specific, localized error and provide a correct fix, allowing the agent to move forward.

Listing 6 The 14B-SR agent successfully uses reflection to fix the specific API error, demonstrating its value as a targeted debugger.

```
# LOG: SR critique after Step 4
The code failed with an AttributeError because the `str` object does not have
a method named `mtable`. The correct method is `maketrans`.

# LOG: Revised code generated after the critique
punc = '!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
transtab = str.maketrans(punc, ' '*len(punc)) # Corrected method name
```

A.2.3 Conclusion of Case Study

This comparison provides strong evidence for the effectiveness of Self-Reflection on a mid-tier model. The baseline agent's catastrophic failure shows that even a 14B parameter model can be unreliable. The SR agent, however, was not only able to correctly interpret the task but also to methodically identify and correct its own implementation errors. This iterative debugging process, guided by the critique-and-revise loop, was directly responsible for its ability to produce a successful, high-scoring submission that the baseline could never have achieved. This supports the conclusion that SR is most valuable for models that are capable enough to produce a reasonable plan but still prone to rectifiable implementation bugs.

A.3 Case Study: The Impact of Self-Consistency (SC)

The Self-Consistency (SC) strategy aims to improve robustness by generating multiple candidate solutions and using execution feedback as a verifier. This case study, comparing the baseline and SC agents on the 14B model for the `aerial-cactus-identification` task, provides a clear example of how this strategy can overcome a model's specific cognitive blind spots.

A.3.1 The Baseline Agent: A Narrative of Unrecoverable Failure

The baseline 14B agent's journey on this task was one of frustrating repetition. It demonstrated a good high-level understanding of the problem (image classification with a CNN) but failed on a single, critical implementation detail: correct file pathing.

Listing 7 The 14B baseline agent got stuck in a loop, repeatedly generating logically sound code that failed due to a single, persistent `FileNotFoundError`. It was unable to debug this simple pathing issue over its entire run.

```
# LOG: A representative code attempt from the 14B baseline agent.
# This pattern was repeated across all 25 steps.

import pandas as pd
from PIL import Image
import torch
# ... other imports ...

# The agent correctly identifies the file with labels.
train_df = pd.read_csv("train.csv") # <-- ERROR: Incorrect path

# The agent defines a correct PyTorch Dataset...
class CactusDataset(Dataset):
    def __init__(self, df, img_dir):
        # ...
    def __getitem__(self, idx):
        img_path = os.path.join(self.img_dir, self.df.iloc[idx]['id']) # <-- This logic
        ↪  is sound
        # ...

# ...but the initial file read always fails.
# TRACEBACK:
# FileNotFoundError: [Errno 2] No such file or directory: 'train.csv'
```

The agent's linear, "depth-first" reasoning process latched onto this single error, and it was unable to escape. Its subsequent debugging attempts involved changing the model architecture or other high-level details, but never addressed the fundamental pathing problem.

A.3.2 The Self-Consistency Agent: A Breakthrough via Parallel Exploration

In stark contrast, the SC agent, by generating three distinct code versions at each step, successfully navigated this challenge. While most of its initial candidates also failed, this parallel exploration dramatically increased the probability of generating a correct solution.

The "Golden Ticket" Solution (Step 5). After four steps of exploring various failures (including `FileNotFoundError`, `NameError`, and `TypeError`), one of the three candidates generated in Step 5 was a complete, end-to-end success.

Listing 8 The successful candidate from the 14B-SC agent. By correctly specifying the `./input/` prefix, it overcame the exact error that doomed the baseline agent.

```
# LOG: The successful candidate code from Step 5 (Node 0d8b49...)
# RESULT: is_buggy: false, Validation Score (AUC): 0.9995

# The agent correctly uses the path for the training labels.
train_df = pd.read_csv("./input/train.csv") # <-- The CRITICAL FIX

# The agent correctly defines the image directory.
img_dir = "./input/train"

# The agent uses a robust PyTorch implementation with a pre-trained ResNet18,
# a custom Dataset, and a correct training and prediction loop.
model = models.resnet18(pretrained=True)
# ...
# ... (successful execution follows) ...
```

Conclusion of Case Study. This comparison provides powerful evidence for the value of Self-Consistency. It demonstrates that for a mid-tier model like the 14B, which has strong latent capabilities but is prone to simple, high-impact errors, SC is a highly effective strategy. It transforms the problem-solving process from a brittle, linear path into a robust, parallel search. The SC agent succeeded not because it was inherently "smarter" on any single attempt, but because it gave itself more opportunities to be "correct," eventually generating a high-quality solution that the baseline agent was fundamentally incapable of reaching.

A.4 Case Study: Planner-Coder on DeepSeek-32B

The Planner-Coder (PC) strategy, which decouples high-level planning from low-level code implementation, was the most effective strategy for the 32B model. This case study from the `spooky-author-identification` competition illustrates how this decomposition prevents "library confusion"—a common and critical failure mode for monolithic agents.

A.4.1 Initial Failure: Library Confusion in Segmented Generation

In its first attempts, the agent's Planner component correctly produced a high-level plan centered around using `scikit-learn`'s `LogisticRegression`. However, the Coder component, when generating the implementation, lost context and defaulted to its strong prior for PyTorch syntax, leading to an inevitable error.

Listing 9 Initial failure of the PC agent. Despite a clear `scikit-learn` plan, the Coder component generated a `PyTorch` training loop in a later segment, creating an incompatible script.

```
# LOG: Planner Output (Step 1)
# Plan:
# 6. *WHAT:* Train a multi-class logistic regression model.
#    *HOW:* Use LogisticRegression(solver='lbfgs', multi_class='multinomial')...

# LOG: Coder Output for the "Modeling" Segment (Correctly follows plan)
# Thought: Instantiating the logistic regression model as specified in the plan.
model = LogisticRegression(solver="lbfgs", multi_class="multinomial", random_state=42)

# LOG: Coder Output for the "Training & Validation" Segment (Incorrect)
# Thought: Defining the loss function and optimizer for the model.
# ERROR: The Coder incorrectly generates PyTorch code here, ignoring the sklearn model.
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

# TRACEBACK:
#   File "runfile.py", line 75, in <module>
#     DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
# NameError: name 'torch' is not defined
```

A.4.2 Eventual Success: Coherent Plan Enables Correct Implementation

After two failed attempts, the agent adapted. In a subsequent step, the Planner produced a new, clear, and architecturally consistent plan that relied exclusively on `scikit-learn`. This provided the Coder with an unambiguous blueprint.

Listing 10 The successful, coherent plan generated in Step 3. It specifies a clear, end-to-end `scikit-learn` pipeline without any conflicting library references.

```
# LOG: Successful Planner Output (Step 3)
## Plan:
# 3. Text Vectorization:
#    *WHAT:* Convert text data into numerical features using TF-IDF.
#    *HOW:* Use tfidf = TfidfVectorizer(...).
# 4. Model Training:
#    *WHAT:* Train a logistic regression model on the TF-IDF features.
#    *HOW:* Use model = LogisticRegression(...).
```

Guided by this coherent plan, the Coder successfully generated a complete, bug-free script where each segment correctly implemented the corresponding plan step, resulting in a successful run. This outcome demonstrates the core benefit of the Planner-Coder strategy: by forcing the agent to first create a high-level, consistent blueprint, it dramatically reduces the likelihood of implementation-level errors.

Listing 11 The successful code generated by the PC agent. Each segment correctly and consistently implements the `scikit-learn` plan, avoiding library confusion.

```
# LOG: Final Successful Code from the PC Agent (Step 3)
# EXECUTION RESULT: Success, Validation Accuracy: 0.8077

# --- Segment: Setup & Imports ---
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
# ... other sklearn imports

# --- Segment: Modeling ---
# Thought: Instantiate logistic regression model as per Master Plan step 4.
model = LogisticRegression(max_iter=1000, multi_class="multinomial", random_state=42)

# --- Segment: Training & Validation ---
# Thought: Fit the logistic regression model on the training data.
model.fit(X_train, y_train)
# ... (evaluation code) ...

# --- Segment: Prediction & Submission ---
# Thought: Get predicted probabilities for test data.
test_probs = model.predict_proba(X_test_tfidf)
# ... (submission file creation) ...
```

Appendix B. Prompt Design for Core Agent Operations

The effectiveness of the agentic strategies implemented in this thesis is highly dependent on the quality and specificity of the prompts used to guide the Large Language Model. This appendix details the principal system prompts used for the core generative operations: drafting, debugging, and improving solutions. These prompts were carefully engineered to elicit the desired behavior from the underlying models.

B.1 Prompt for Initial Solution Drafting

This prompt is used when the agent is tasked with creating a new solution from scratch. It frames the model as a "Kaggle Grandmaster" and provides highly detailed instructions on the expected output format, which must include both a step-by-step `PLAN` and the corresponding `CODE`.

Listing 12 The system prompt used for the solution drafting operation.

```
SYSTEM: |
  You are a Kaggle Grandmaster. Your task is to devise a clear, step-by-step
  PLAN and then write the corresponding Python CODE to solve machine learning
  competitions. Adhere strictly to the specified output format. The primary
  goal for this draft is a working, bug-free solution, so prioritize
  simplicity and correctness in your design.
user_instructions:
  Task Context / Possible Questions: You will be provided with a description of
    a Kaggle competition and asked to generate a complete solution...
  How to Answer (Output Structure, Plan Details, Code Details, Examples): |
    Your entire response MUST be structured in two main sections: 'PLAN:'
    followed by 'CODE:'. Use '---' as a separator...

    **1. PLAN Section Requirements:**
       Construct a "PLAN:" section. This plan must consist of 7-10 highly
       detailed, sequential bullet points. Each point must describe a specific,
       actionable step... (e.g., "1. Data Loading: Load `train.csv`...")

    **2. CODE Section Requirements:**
       Follow the PLAN with a "CODE:" section, containing a single, complete
       Python script... Crucially, before every distinct logical block of code...
       you MUST include a comment starting with "# Thought:"...
  Critical Adherence / Final Instructions: |
    Strict adherence to the detailed PLAN structure... and the '# Thought:'
    commenting convention... is mandatory. The primary objective for this draft
    is a working, bug-free solution...
```

B.2 Prompt for Decomposed Task Generation (Planner)

Listing 13 The system prompt for the "Planner" role in the Decomposed Task Generation strategy.

```
SYSTEM: |
  You are an expert Kaggle Grandmaster and a meticulous Technical Lead. Your primary
  responsibility is to create an exceptionally detailed, actionable, and high-quality
  strategic Master Plan for solving a given machine learning competition... Your
  output MUST be a 'Task Summary' followed by a 'Plan'. You do NOT write any code.

user_instructions:
  Input Understanding: You will receive: a 'Full Kaggle Competition Description',
    a 'Data Overview', and potentially a 'Memory of Previous Attempts'.
  Output Requirements (Strict Adherence Mandatory): |
    Your entire response MUST strictly follow this two-part structure...

    ## Task Summary:
       - Provide a concise summary (around 5-7 sentences)...

    ## Plan:
       - Construct a list of 7-10 sequential, numbered bullet points...
       - **Crucial Detail for Each Plan Step (WHAT, HOW, WHY):** For *every*
         bullet point, you MUST explicitly detail:
           a. **WHAT** is the specific action...
           b. **HOW** this action will be achieved: Be extremely specific. Mention key
              Python libraries (e.g., `pandas`, `sklearn.impute`), specific
              functions/classes (e.g., `pd.read_csv()`, `SimpleImputer()`)...
           c. **WHY** this step is necessary...
  Critical Reminder: Your role is exclusively planning and summarizing... The Coder
    agent relies ENTIRELY on the clarity, explicitness... and logical correctness of
    your 'Task Summary' and 'Plan'. DO NOT generate any Python code.
```

B.3 Prompts for Self-Reflection Agent

The Self-Reflection strategy uses a two-step process, with a distinct prompt for each stage to guide the LLM into the correct role.

B.3.1 Critique Stage ("Reviewer" Role)

Listing 14 The system prompt for the "Reviewer" role in the Self-Reflection strategy.

```
SYSTEM: |
  You are a senior data scientist, trying to check a code written by a junior
  data scientist to solve a machine learning engineering task - a Kaggle competition.
How to answer the user: |
  Whenever you answer, always:
  1. Write a "Review" section in plain text-explaining the main mistake(s).
  2. Then write an "Instructions" section: a NUMBERED list of fix instructions.

# The user message then contains the full context:
# {
#   "Question": "I am writing a code to solve this task: {task_desc}...",
#   "Code to Review": "{code}",
#   "Execution Output": "{term_out}",
#   "Execution Feedback": "{analysis}",
#   "Your Task": "Provide a Code review for mistakes and bugs...",
#   "Rules I need you to follow": "RULE 1: **DO NOT WRITE ANY PYTHON CODE...**"
# }
```

B.3.2 Revision Stage ("Coder" Role)

Listing 15 The system prompt for the "Coder" role in the Self-Reflection strategy.

```
SYSTEM: |
  You are a Kaggle Grandmaster and a precise coder. You will receive a Kaggle
  competition code, a review on the correctness of this code, and Instructions
  on how to improve its correctness.
Task: |
  Your task is to help your team win the competition by following the code review
  and implementing the suggested instructions.
How to reply to the user: |
  Whenever you answer, always:
  1. Think of a "Plan:" section... Wrap this section between <think></think> tags...
  2. Then write a Revised Code: section titled '# Applying edits based on review.'
     containing the full new improved code...
CRITICAL REQUIREMENT: |
  Always make sure that you don't provide partial solutions; your final code
  block should be complete, starting from imports, until saving the submission.

# The user message then contains the full context:
# {
#   "Question": "I am trying to improve my code...",
#   "Original Code": "{code}",
#   "Edit Instructions": "{reflection_plan}",
#   "Rules": "RULE 1: Apply the steps from 'Edit Instructions'.\nRULE 2: **Do NOT change
↪  any other part of the code...**",
#   "Output Format": "Line 1: Start IMMEDIATELY with a comment '# Applying edits...'
#                     Next Line: Start the Python code block immediately..."
# }
```