LEARNING TO WIN: A PROCESS REWARD MODEL FOR COMPETITIVE MACHINE LEARNING AGENTS

By

Sajjad Ahmed
20366018

A thesis submitted to the Department of Computer Science
and Engineering in partial fulfillment of the requirements
for the degree of Master of Science in
Computer Science and Engineering

Department of Computer Science and Engineering
BRAC University
March 2026

Declaration

It is hereby declared that:

1. The thesis submitted is my/our own original work while completing degree at Brac University.
2. The thesis does not contain material previously published or written by a third party, except where this is appropriately cited through full and accurate referencing.
3. The thesis does not contain material which has been accepted, or submitted, for any other degree or diploma at a university or other institution.
4. We have acknowledged all main sources of help.

Author:

Sajjad Ahmed
20366018

Supervisor:

Md. Golam Rabiul Alam, PhD
Professor, Department of Computer Science and Engineering
BRAC University
Email: rabiul.alam@bracu.ac.bd

Approval

The thesis titled *Learning to Win: A Process Reward Model for Competitive Machine Learning Agents*, submitted by Sajjad Ahmed (Student ID: 20366018), has been accepted as satisfactory in partial fulfillment of the requirements for the degree of M.Sc. in Computer Science and Engineering at BRAC University.

Supervisor

Md. Golam Rabiul Alam, PhD
Professor, Department of Computer Science and Engineering
BRAC University

External Examiner

Dr. Chowdhury Farhan Ahmed, PhD
Professor, Department of Computer Science and Engineering
University of Dhaka

Internal Examiner

Dr. Muhammad Iqbal Hossain
Associate Professor, Department of Computer Science and Engineering
BRAC University

Program Coordinator

Dr. Md Sadek Ferdous
Professor, Department of Computer Science and Engineering
BRAC University

Chairperson

Sadia Hamid Kazi, Ph.D.
Associate Professor, Department of Computer Science and Engineering
BRAC University

Ethics Statement

This research uses publicly available competition datasets (Kaggle) together with anonymized agent human feedback JSON logs collected for this study. No private personal data was collected directly from human participants. All sources were used in accordance with their respective platform terms and citation requirements.

Abstract

Large Language Model (LLM)-based coding agents are increasingly deployed for multi-step data science tasks, yet no systematic study has examined how these agents behave across diverse competitive problems. This thesis presents a two-part investigation using 194 human-supervised agent human feedback JSONs across 103 Kaggle competitions (1,584 turns total). First, we conduct a comprehensive empirical study. We also propose and show that process-oriented quality metrics. Second, we develop a Process Reward Model (PRM) for data science workflows—the first application of PRMs beyond mathematical reasoning—that predicts whether an intermediate step will improve competition scores. Using 49 tabular features across 6 feature groups, we train XGBoost and LightGBM baselines and conduct a full feature ablation study. Code features (code length, imports, deltas) emerge as the strongest individual tabular predictors. We further fine-tune Qwen2.5-Coder-3B with QLoRA as a hybrid language-model-based PRM (v2), incorporating code snippets and tabular features into semantic prompts, which achieves AUROC 0.656—surpassing the best tabular baseline (0.530) by 12.6 points—by leveraging semantic content from agent plans, reflections, and code. To establish the necessity of domain-specific fine-tuning, we evaluate zero-shot frontier models (i.e., GPT-5, Gemini 3.0 Flash/Pro, Llama 3.3 70B) on the same task. GPT-4o achieves the highest zero-shot AUROC (0.709), outperforming our fine-tuned 3B model, while most other frontier models fall to majority class prediction (AUROC 0.500). This demonstrates that while powerful zero-shot models can reason about the task, domain-specific fine-tuning remains valuable for smaller models, achieving competitive performance with 200x fewer parameters. Ultimately, our research demonstrates that observable code signals are significantly more reliable than an agent's articulated plans for predicting success, providing a robust foundation for the development of real-time guidance systems in competitive data science.

Keywords: Large Language Model(LLM), Agents, Process Reward Model, Reinforcement Learning from Human Feedback (RLHF), Data Science, Competitive Machine Learning

Acknowledgement

I would like to express my sincere gratitude to my thesis supervisor, Md. Golam Rabiul Alam, PhD, for providing invaluable guidance and support throughout this research. His expertise and encouragement was instrumental in shaping this work. I am grateful to the Department of Computer Science and Engineering at BRAC University for providing the resources and academic environment necessary to conduct this research. I would also like to thank the contributors who generated the human feedback JSON dataset used in this study, as well as the broader open-source machine learning community whose tools and frameworks made this work possible. Finally, I thank my family and friends for their unwavering support and encouragement throughout my academic journey.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Large Language Models (LLMs) are no longer used only as assistants that produce snippets of code on demand. Increasingly, they are embedded in agent systems that operate over many steps: reading a task, forming a plan, writing and running code, reacting to errors, and iterating until a measurable goal is reached. This shift matters because multi-step work is where most real-world engineering and data science effort lives.

Competitive data science is a particularly clear setting to study these agents. A Kaggle-style workflow forces an end-to-end pipeline—data ingestion, cleaning, modelling, evaluation, and submission—and provides an objective signal of success in the form of a leaderboard score. At the same time, the path to a good score is rarely linear: progress depends on a sequence of decisions, and many reasonable-looking steps do not help (or help only after several iterations).

1.1 Background

Data science projects require more than correct syntax and boilerplate code generation. Even when the underlying programming is straightforward, success hinges on planning and decision-making: choosing what to explore, what to ignore, which baselines to establish, how to validate, and when to invest in feature engineering or model changes. In practice, the hard part is often not writing code, but deciding *which* code to write next given partial evidence and limited time.

This makes data science agent evaluation meaningfully different from typical software engineering benchmarks. In many software tasks, the objective is to produce a functionally correct patch against a relatively stable specification. In competitive machine learning, the specification is underdetermined: the dataset is the specification, and the objective is comparative performance. The feedback loop is also delayed and noisy—you may only see the effect of a change after a full training run and a submission. As a result, an agent can generate plausible code for many turns and still fail to converge on a strong solution.

1.2 Motivation

These challenges motivate two complementary needs. First, we need empirical evidence about how LLM agents behave in the wild: which strategies they cycle

through, what failure modes dominate, and what differentiates runs that eventually reach meaningful scores. Benchmarks such as SWE-bench and MLE-bench have made progress on realistic evaluation for coding and ML engineering agents [11], [17], but competitive data science remains under-explained at the level of *process*. Second, we need a way of guiding us through the steps in a data-science workflow, which should be one that is smooth. There are Process Reward Models (PRMs) that are a good fit: instead of only scoring the final answer, a PRM is able to grade the intermediate steps [19]. Applying this to data science is new and tricky because the 'ground truth' is only revealed at the end (leaderboard points) and it's entangled up with a bunch of other variables. Still, if a PRM can pick up even a faint but reliable preference for steps that usually lead to better scores, it is a handy tool: It allows the agents to rank possible next moves, to skip the circular loops and focus on the changes that really make a difference.

1.3 Research Questions

Based on these gaps, this thesis addresses the following research questions:

- **RQ1:** What common strategies, failure modes, and success factors appear in the logs of LLM agents working on Kaggle competitions?
- **RQ2:** Can a small Process Reward Model accurately predict if an intermediate agent step will improve the final competition score?
- **RQ3:** Does using a PRM to select the best step lead to better performance on new competitions compared to standard agent generation?

1.4 Contributions

This thesis offers four main contributions to the study of machine learning and autonomous agents:

1. **Strategy classification grounded in real human feedback JSONs:** In this paper, we build a taxonomy of agent strategies and determine how strategies are played out in 194 human feedback JSONs from 103 Kaggle competitions. Instead of simply observing aggregate amounts, we also monitor changes in the strategies over time and associate repeating patterns (e.g. long loops of exploration) with downstream effects.
2. **Curated human feedback JSON dataset for this study:** We compile and clean 194 anonymized agent human feedback JSONs across 103 Kaggle competitions, aligning multi-turn traces, execution logs, and competition scores, and joining available human QA annotations. This structured dataset enables reproducible analyses within the thesis.
3. **A PRM formulation tailored to data science workflows:** We're making that PRM idea, turning it into a setup where feedback is delayed, where the success is a little relative. We define these turn–level labels by the deltas that we observe in the score and model on an attempt to predict whether it's likely

that an intermediate step will come before a score bump based on stuff from the code, strategy, execution and context.

4. **A practical evaluation framework for step selection:** We present a retrospective Best-of-N style evaluation which assesses the ability of PRM to rank candidate steps, pitting it against random choice, PRM-led choice and an oracle upper bound. This framework is a tangible connection between the off-line analysis and deployable guidance in the agent loop iterations.

1.5 Thesis Outline

The rest of this thesis is organized as follows:

- **Chapter 2: Related Work**—Reviews the broader literature on autonomous agents, process supervision, and multi-agent frameworks.
- **Chapter 3: Background Studies**—Explains the foundational concepts of alignment (RLHF, DPO), Process Reward Models (PRMs), and the competitive data science environment used in this research.
- **Chapter 4: Dataset Description and Extraction Pipeline**—Explains how we collected, processed, and labeled the agent log data.
- **Chapter 5: Empirical Study**—Analyzes how the agents behaved, answering RQ1.
- **Chapter 6: Process Reward Model**—Describes the design and training of our PRM, addressing RQ2.
- **Chapter 7: Evaluation**—Shows the test results and simulated deployments, answering RQ3.
- **Chapter 8: Discussion, Limitations, and Future Work**—Summarizes what these results mean, notes the study's constraints, and suggests next steps for research.
- **Chapter 9: Conclusion**—Provides a final summary of the project and its findings.

Chapter 2

Related Work

This chapter is a survey of the recent developments in autonomous agents, process supervision, and multi-agent frameworks putting our contribution in the context of the larger picture of computational research. We give preference to the results of peer-reviewed periodicals and significant conferences to create a stringent theoretical framework.

2.1 LLM Agents for Data Science

There has been a fast development of the use of Large Language Models (LLMs) as autonomous agents in the field of data science. A comprehensive survey by Chen et al. [26] categorizes these agents by their role in the data lifecycle, distinguishing between design-centric agents that plan workflows and execution-centric agents that generate code. Their discussion indicates that a big change has occurred in single-turn code generation to multi-step problem solving where agents have to maintain the state, deal with errors and incorporate domain knowledge. Similarly, the DataWiseAgent framework [27], presented at EMNLP 2025, introduces a notebook-centric architecture that treats data science tasks as state transitions. This enables more powerful long-horizon planning than in the case of earlier, stateless conversation models.

2.2 Process Reward Models and Supervision

Although outcome-based supervision has been the primary generator of reinforcement learning, more recent observations have shifted their focus to Process Reward Models (PRMs) which consider intermediate progressions. Lightman et al. [19] has shown the effectiveness of PRMs when it comes to mathematical reasoning and demonstrated that step-by-step verification essentially minimizes the number of logical errors. Continuing on the topic of this to code generation, Wang et al. [24] suggested the Outcome Refining Process Supervision (ORPS) framework. Process and outcome rewards are merged in their work with the help of executable verification which is used to self-criticize generated code by means of tree-structured search. This technique brought significant enhancements to code correctness and efficiency, indicating dense and execution-sensitive feedback is better than sparse outcome feedback in tasks of complex engineering.

This was further formalized by Ma et al. [20] as Process Supervision-Guided Policy Optimization. The limited nature of rewards in reinforcement learning is addressed in their research through the use of PRMs to give a line-level feedback in the generation of code. These dense rewards were added to the value function initialisation to enable them to perform better in long-horizon problems, which supports the hypothesis that multi-step reasoning requires granular evaluation.

2.3 Multi-Agent Frameworks and Orchestration

The multi-agent architectures that separate responsibilities have also been encouraged by the complexity of data science workflows. Huang et al. [15] came up with AgentCoder, which divides the issues into dedicated roles that include: a programmer, a test designer, and a test executor. This specialization of roles enables refinement loops of a high degree of iteration, which are hard to accomplish with one big agent. The MARSYS platform [25] , similarly, offers an orchestration layer to manage teams of specialized agents, as in the case of MARSYS platform. By specifying formalised collaboration procedures between research, analysis and verification agents, MARSYS proves the claim that modular systems can more reliably support more complex work processes than baseline single agents.
All of these developments lead to a scenario in the future where autonomous data science is motivated by specialised, process-sensitive agents with the ability to self-correct and to plan in the long term. We base our work on such foundations and present a specialized PRM of competitive data science, the gap between domain-independent reasoning and domain-dependent execution evaluation.

Chapter 3

Background Studies

This chapter elaborates the conceptual, methodological and platforms upon which our research is based. We describe the alignment algorithms, process reward model architecture, and the data science environment (competitive) of evaluation.

3.1 Alignment and Preference Learning

Traditionally, language models are trained to predict the next token, although such is not a goal that ensures helpfulness or safety. Alignment methods are used to direct the model behavior in accordance with the human intent.

3.1.1 Reinforcement Learning from Human Feedback (RLHF)

Standard RLHF [2], [5] involves three steps: (1) supervised fine-tuning (SFT) of an underlying model; (2) training a reward model (RM) on human preference pairs (between better and worse responses); and (3) the policy is optimized by using Proximal Policy Optimization (PPO) against this reward model. Although effective, RLHF is computationally expensive and can be hacked [6] during rewards provision.

3.1.2 Direct Preference Optimization (DPO)

Direct Preference Optimization (DPO) by Rafailov et al. [10] was developed to deal with the complexity of PPO. DPO is an implicit maximization of the reward function and consists in considering the language model as the reward model. It reduces a classification loss on preference pairs, avoiding the existence of a distinct reward model and a reinforcement learning loop. Azar et al. [7] with IPO and Ethayar et al. [12] with KTO have expanded this method to enhance stability and data efficiency, respectively.

3.1.3 Scalable Supervision (RLAIF)

Due to the inability of human annotation to scale, AI feedback (RLAIF) [4], [9]has become a target of research. In this case, the preferences/critiques are created based on a powerful teacher model, and they are fed into the target model. Our work is based on this technique, where we do not use large-scale human labeling, but automated signals (competition scores and automated judges).

3.2 Process Reward Models (PRMs)

Single scalar score of a final output is offered by standard reward models (Outcome Reward Models, or ORMs). Process Reward Models (PRMs) on the other hand score intermediate reasoning in between. Mathematically, given a human feedback JSON $\tau = (s_1, a_1, s_2, a_2, \ldots, s_T)$, a PRM estimates the value $V(s_t)$ or quality $Q(s_t, a_t)$ of each step. Lightman et al. [19] demonstrated that this granular feedback is important in multi-step reasoning tasks such as mathematics, where one initial error can annul the long sequence of correct subsequent steps.
While most prior PRM work targets domains with relatively explicit and verifiable intermediate correctness (e.g., mathematical reasoning), competitive machine learning and data science workflows differ in three ways. First, intermediate steps such as exploratory analysis, data cleaning, feature engineering, or hyperparameter tuning are rarely "correct" or "incorrect" in isolation; their value is revealed only through delayed outcomes (training curves, validation metrics, and ultimately a scored submission). Second, the state is intrinsically *multi-modal*: beyond natural language plans and reflections, the decisive signals are often contained in executable artefacts (code snippets, library imports, execution success, runtime, and observed score deltas). Third, evaluation is relative and task-specific: what improves one competition can regress another, so a useful PRM must learn preferences that generalise across heterogeneous competitions while remaining sensitive to context such as submission budgets and score directionality. These properties motivate a domain-specific PRM formulation for competitive machine learning workflows, where the model consumes both text and code-derived features and predicts whether a step is likely to precede measurable score improvement.

3.3 Competitive Data Science Environment

We assess the agents in the framework of Kaggle competitions [18]. There are specific challenges in this platform to the typical software engineering benchmarks:

- **Objective Metrics:** Success is defined by a specific scoring metric (e.g., RMSE, LogLoss, Accuracy) calculated on a hidden test set.
- **Underspecified Tasks:** Contrary to unit-test-driven development it is not known what the correct solution is. The feature space and model architectures should be investigated in order to identify the best solution.
- **Resource Constraints:** Solutions are constrained by runtime limits (typically 9 hours) and hardware availability (GPU/TPU quotas).

3.4 Agent Architectures

We are using the MLE-bench framework as an implementation of the data science agent loop formalization in the form of MLE-bench framework [11]. The agent runs in a read evaluate print loop (REPL):
1. **Observation**: The agent scans the description of competition, file system, and the logs of the past executions.

2. **Thinking**: It devises a course of action and contemplates the past results.
3. **Action**: It creates code (Bash or Python) to run.
4. **Execution**:The code executes in a sandbox environment giving back either stdout/stderr or competition score.

This construction cycle reflects the ReAct (Reason+Act) paradigm, with the opportunity of self-correcting and making dynamic changes depending on feedback.

Chapter 4

Dataset Description and Extraction Pipeline

This chapter describes the data that we employed in our empirical study. It includes the source of the agent logs, how we retrieved the data and how we verified it, as well as the structure of the end result, the dataset. This data is used to analyse in Chapter 5 to Chapter 7.

4.1 Data Sources

4.1.1 Human feedback JSON Data

The primary dataset refers to 194 cleaned JSON files of logs of the agents. The last dataset contains 194 distinct agent execution that we refer to as human feedback JSONs after the elimination of duplications and confirmation that there are no lost instances. These paths represent 103 various competitions at the Kaggle platform [18].

Human feedback quality controls

To assess and improve the quality of the human-supervised traces used in this thesis, we applied quality controls at three levels:

- **Structural validation and cleaning:** we removed duplicates, verified that each human feedback JSON was complete (no missing turns/artifacts), and excluded corrupted or partially logged runs. This ensures that downstream feature extraction (plans, reflections, code, execution logs, and scores) is consistent.
- **Expert review signals (QA reports):** for 79 trajectories, human reviewers provided turn-level QA reports across multiple dimensions (Table 4.1). These scores provide an explicit record of expert judgement on technical soundness, dataset usage, and reflection–code alignment, and allow us to analyse where human-perceived quality diverges from measured score outcomes (Chapter 5).
- **Validation of derived annotations:** because some labels in our pipeline (e.g., strategy intent) are algorithmically inferred, we performed manual spot-

check validation on a random sample of turns (Section 4, Methodological Validation) to verify that derived labels are reasonable and to characterise expected noise.

Importantly, we retain both successful and failed trajectories in the dataset to support a realistic analysis of agent failure modes (e.g., exploration loops and no-submission runs), rather than filtering the corpus to only high-scoring outcomes.

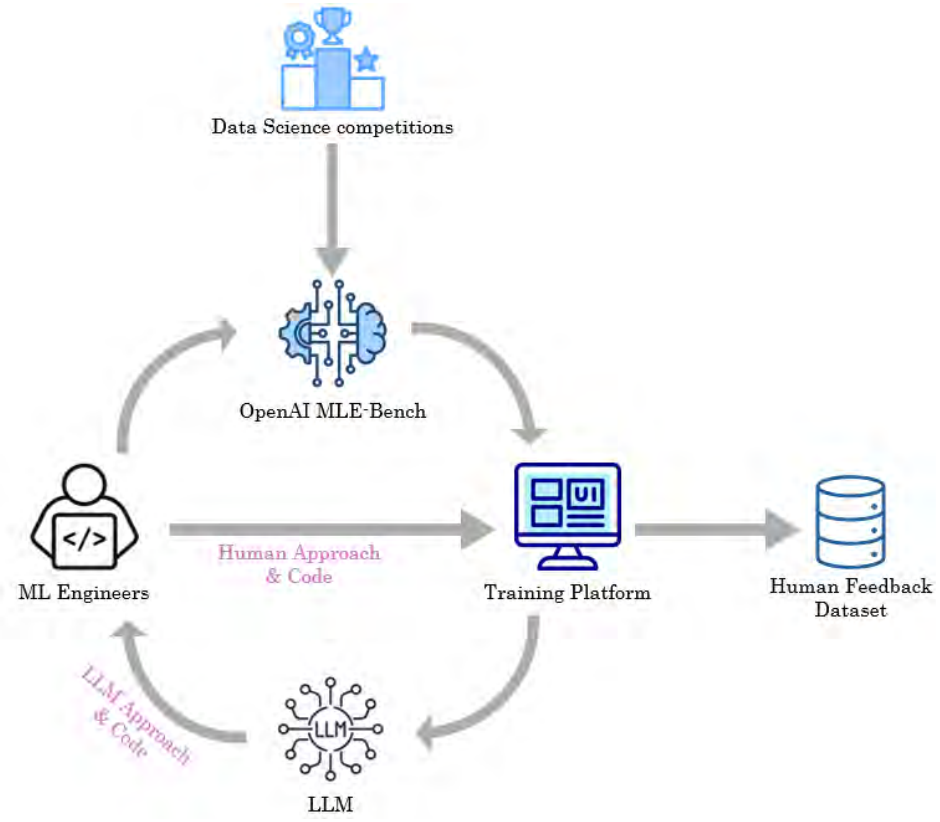

Figure 4.1: Data preperation process.

Overview of the RLHF Data Collection Pipeline

This Fifure in 4.1 illustrates a structured data collection pipeline designed to generate a high-quality human feedback dataset (typically formatted in JSON) for Reinforcement Learning from Human Feedback (RLHF). The primary goal of this process is to align a Large Language Model (LLM) with expert-level problem-solving in machine learning tasks.
The workflow can be broken down into the following key stages:

- **Task Curation & Benchmarking:** The pipeline originates with real-world, complex problems sourced from Data Science competitions. These challenges are formalized to populate the OpenAI MLE-Bench (Machine Learning Engineering Benchmark), providing a rigorous set of tasks for evaluation.
- **Model Generation:** The target LLM attempts the benchmarked tasks, generating an initial proposed solution, denoted in the diagram as the *LLM Approach & Code*.

- **Expert Human Evaluation:** This model-generated output is routed to expert ML Engineers. The engineers review, evaluate, and potentially correct the model's work.
- **Feedback Aggregation:** The ML Engineers then submit their refined, gold-standard solutions and evaluations—the *Human Approach & Code*—directly into a centralized Training Platform. The Training Platform also orchestrates the distribution of the MLE-Bench tasks to both the LLM and the engineers.
- **Dataset Compilation:** Finally, the Training Platform compiles the benchmark tasks, the model's initial outputs, and the expert human corrections into a structured Human Feedback Dataset. This dataset serves as the foundational truth for RLHF, used to fine-tune the LLM to write better code and approach ML engineering problems more accurately.

Every human feedback JSON captures a protracted engagement in which an agent solves a data science issue. The process works like this:

- The description of a Kaggle competition and the corresponding dataset is given to the agent.
- In every turn, the agent provides a structured response which consists of:
  - **Analysis**: A review of the dataset and the problem goals.
  - **Plan**: A step-by-step list of what the agent wants to do next.
  - **Proposed Code**: Python code written to carry out the plan.
  - **Reflection**: The agent's own check of whether its plan and code make sense.
  - **Revised Plan**: A new plan, created if the agent notices an error during reflection.
  - **Finalized Code**: The code the agent actually decides to run.
- A safe, isolated environment runs the finalized code. The system records the standard output, any error messages, and how long the code took to run.
- When the agent submits a final prediction, a scoring system returns the competition score and indicates if the score qualifies for a bronze, silver, or gold medal.

4.1.2 Quality Assessment (QA) Reports

We also included 79 human-written Quality Assessment (QA) reports. Human reviewers read the agent's outputs and graded them. Reviewers scored each turn on a scale from 1 to 5 across twelve quality dimensions, detailed in Table 4.1.

4.1.3 Dataset Statistics

Table 4.2 shows the main numbers for our final, cleaned dataset.

Table 4.1: Quality Assessment (QA) dimensions and evaluation criteria.

Dimension	**Criteria**
`technical_soundness`	Correctness and idiomatic use of code, methods, and libraries.
`reflection_quality`	Depth, insight, and coverage of self-critique regarding the plan and execution.
`dataset_usage`	Appropriate data handling, including correct splits and avoidance of leakage.
`instruction_following`	Adherence to user constraints, format requirements, and workflow instructions.
`plan_reflection_consistency`	Logical alignment between the stated plan, the reflection, and the revised plan.
`code_output_consistency`	Match between the described intent/expected output and the actual code behavior.
`factuality`	Accuracy of statements made about the code, data, or results.
`reflection_code_alignment`	Whether the reflection accurately describes the implemented code.
`human_edit_coherence`	Consistency of human edits with the original agent intent (for edited turns).
`assistant_generation_integrity`	Completeness of the response, including proper formatting and lack of truncation.
`gpt_provenance`	Absence of specific "As an AI" refusals or identity disclosures.
`data_leakage`	Explicit check for use of test data during training or validation.

Table 4.2: Dataset statistics after removing duplicates and validating the files.

Metric	**Value**
Total conversational turns	1,584
Unique agent human feedback JSONs	194
Unique Kaggle competitions	103
Unique human annotators	24
Turns yielding competition scores	268
Turns accompanied by QA data	550
Average turns per human feedback JSON	8.2
Human feedback JSONs concluding with scores	144 (74.2%)
Human feedback JSONs failing or unscored	50 (25.8%)

4.1.4 Competition Typology Distribution

To classify the Kaggle competitions by the keywords in the competition titles we divided them into the following four main categories: **Tabular** (predicting numbers or categories based on the spreadsheets), **Image** (processing images or medical scans), **NLP** (text processing or question-answering), and **Audio** (speech recognition or sound recognition). Table 4.3 shows how these types are split across the competitions, human feedback JSONs, and individual turns.

Table 4.3: Distribution of competition types. Tabular data problems make up the majority of the dataset.

Typology	**Comp.**	**%**	**Traj.**	**%**	**Turns**	**%**
Tabular	62	60.2	96	49.5	798	50.4
Image	25	24.3	50	25.8	368	23.2
NLP	14	13.6	45	23.2	396	25.0
Audio	2	1.9	3	1.5	22	1.4
Total	103	100	194	100	1584	100

Tabular competitions comprise the largest portion of our data (60% of the unique competitions, 50% of the total turns). Image tasks are the second most common with 24%. While NLP tasks only constitute 14% of the competitions, they constitute 25% of the turns. This means agents usually took more steps to complete text-based problems. Audio tasks are very rare in this dataset with only 2 competitions included.

4.2 Extraction and Parsing Pipeline

We wrote a Python script to get the data out of the raw JSON logs and QA reports. The script turns all this information into one CSV file, with 47 data columns for each turn. The steps of this extraction are provided below.

4.2.1 Score Resolution and Normalization

The script looks at several places in the JSON file in order to find the competition score for a particular turn:

1. `execution.score`
2. `execution.output.test_fitness`
3. `execution.output.grading_report.score`

After finding the scores, the script is used to calculate the difference, or delta, between each turn. Because some competitions care about getting a high score (when it comes to accuracy, for example) and others want a low score (when it comes to the rate of errors), the script changes these values so that a positive delta will always mean that the agent's performance improved.

4.2.2 Strategic Intent Classification

In order to understand what the agent is doing at each step we came up with a rule-based classifier. This tool examines the words that the agent speaks and determines which strategy category the turn belongs to one of nine strategy categories. If the tool is not able to categorise the text, it categorises the turn as `unknown`. The categories are defined in Table 4.4.

Table 4.4: Strategy categories and definitions used in the rule-based classifier.

Strategy	**Description**
`exploration`	Initial data inspection, checking file structures, shape, info, and description.
`preprocessing`	Cleaning data, handling missing values, encoding categorical variables, and scaling features.
`feature_engineering`	Creating new features, transformations (log, polynomial), and dimensionality reduction.
`baseline_model`	Training simple initial models like Logistic Regression or Dummy Classifiers.
`model_upgrade`	Switching to more complex models (e.g., XGBoost, LightGBM, Neural Networks).
`hyperparameter_tuning`	Optimizing model parameters using grid search, random search, or frameworks like Optuna.
`ensemble`	Combining multiple models via stacking, blending, or voting.
`debugging`	Fixing code errors, handling exceptions, and troubleshooting runtime failures.
`submission`	Generating and saving final prediction files for competition submission.

4.2.3 Error Taxonomy

In situations where the code of the agent does not execute, the script will read the error message and place it under a standard error category which will be defined in Table 4.5.

4.3 Methodological Validation

In order to ensure that our strategy classifier was functioning properly we randomly picked out 194 turns and verified them manually. We ensured that this random sample had the same distribution of strategy labels and competition types as in the main dataset. By hand checking these, we were able to verify that we were correct in our rules and in cases where any mistake the script was making was constant.

Table 4.5: Taxonomy of execution errors identified in agent traces.

Error Type	**Description**
`timeout`	Execution exceeded the allowed time limit.
`memory_error`	Out of memory (OOM) or CUDA out of memory errors.
`syntax_error`	Code parsing failures due to invalid Python syntax.
`import_error`	Failures to import modules or missing dependencies.
`type_error`	Operations applied to inappropriate data types.
`value_error`	Operations receiving arguments of correct type but inappropriate value.
`key_error`	Attempts to access non-existent dictionary keys.
`index_error`	Attempts to access list indices out of range.
`runtime_error`	Generic runtime exceptions not covered by other categories.
`other_error`	Unclassified or miscellaneous errors.

4.4 Final Output Artifact

The extraction script outputs a single CSV file containing data for all 1,584 turns, with 47 specific features recorded for each turn. We use this CSV file for all the tests and modeling in the rest of the thesis. It is the basis for the behavioral study in Chapter 5, the Process Reward Model in Chapter 6, and the final evaluations in Chapter 7.

Chapter 5

Empirical Study

This chapter presents an empirical analysis of LLM agent behaviour in competitive data science. We analyse strategy distributions, outcome patterns, transition dynamics, score progression, failure modes, cross-annotator variability, and the relationship between question–answer (QA) quality metrics and competition scores. The findings address RQ1: *What strategy patterns, failure modes, and success factors characterize LLM agent human feedback JSONs in competitive data science?*

5.1 Strategy Distribution

We analysed 1,583 annotated strategy steps across the collected human feedback JSONs. Table 5.1 presents the distribution of strategy types observed in the dataset.

Table 5.1: Distribution of strategy types across all annotated steps.

Strategy	**Count**	**Percentage**
exploration	521	32.9%
submission	283	17.9%
preprocessing	215	13.6%
hyperparameter_tuning	151	9.5%
unknown	114	7.2%
baseline_model	83	5.2%
model_upgrade	79	5.0%
debugging	51	3.2%
feature_engineering	44	2.8%
ensemble	43	2.7%
Total	**1,583**	**100.0%**

Exploration dominates the strategy distribution at 32.9%, followed by submission (17.9%) and preprocessing (13.6%). Modelling-related strategies (baseline_model, model_upgrade, ensemble) together account for 12.9%, while hyperparameter_tuning and feature_engineering represent 9.5% and 2.8% respectively. The prominence of exploration reflects the iterative, exploratory nature of competitive data science workflows.

5.2 Outcome Distribution

Table 5.2 summarises the distribution of competition outcomes across 194 human feedback JSONs. We obtained competition-level thresholds (`gold_threshold`, `silver_threshold`, `bronze_threshold`, `median_threshold`) from two sources: (1) the `grading_report` inside the `execution.output` of submit turns for 17 competitions, and (2) Kaggle API leaderboards for 69 additional competitions, where we calculated thresholds as top 10% (gold), top 20% (silver), top 40% (bronze), and median (50th percentile). For each human feedback JSON, we iterate over every turn where `tool_type` is `"submit"` and compare `execution.score` against these thresholds, accounting for score direction (`is_lower_better`). When a human feedback JSON contains multiple submissions, we assign the highest achievement using the precedence: gold > silver > bronze > above_median. This methodology provides threshold coverage for 176 out of 194 human feedback JSONs (90.7%).

Table 5.2: Distribution of competition outcomes across human feedback JSONs. Medal classifications are determined by comparing `execution.score` against competition thresholds from MLE-bench grading reports and Kaggle API leaderboards; the highest medal across all submissions in a human feedback JSON is used.

Outcome	**Count**	**Percentage**
below_median	106	54.6%
no_score	50	25.8%
gold	19	9.8%
above_median	14	7.2%
bronze	3	1.5%
silver	2	1.0%
Total	**194**	**100.0%**

Having both grading report and Kaggle API data on the threshold, we find that 19.6% of the human feedback JSONs were performing within the medal range (gold, silver, bronze, or above-median), and 54.6% were below the median threshold. One quarter (25.8%) got no score whatsoever. It is worth noting that 19 human feedback JSONs (9.8 percent) had gold medals with person 19 and person 11 respectively having 5 and 3 gold medals respectively. This focus indicates that the knowledge of annotators has a significant effect on agent performance. The gold-medal competitions span diverse domains including computer vision (aerial-cactus-identification, denoising-shabby-pages, dogs-vs-cats-redux, invasive-species-monitoring, leaf-classification, plant-pathology-2020), natural language processing (detecting-insults-in-social-commentary, linking-writing-processes-to-writing-quality, lmsys-chatbot-arena), and tabular prediction (netflix-appetency, nomad2018-predict-transparent-conductors, tabular-playground-series-feb-2022). Two human feedback JSONs achieved silver medals and three achieved bronze. Figure 5.1 visualises this distribution.

Medal outcomes are determined by comparing submission scores against competition thresholds from MLE-bench grading reports and Kaggle API leaderboards. This skewed distribution highlights the difficulty of achieving strong performance in competitive data science, even with LLM assistance.

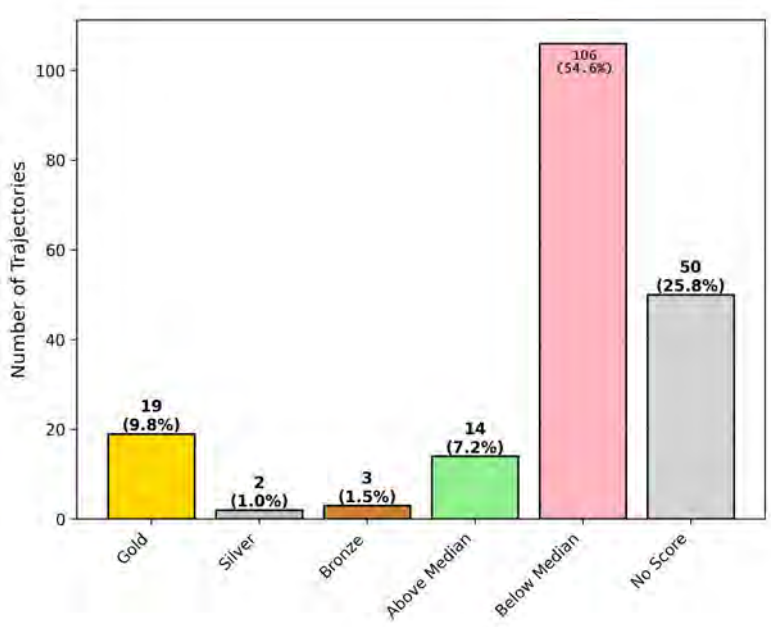

Figure 5.1: Outcome distribution across 194 agent human feedback JSONs.

5.3 Strategy Transition Analysis

On examining the sequence of the agents strategies, we are able to find apparent differences between a successful and a failed run. A successful agent typically follows a rational sequence of actions: It has to start by scanning the data (exploration), cleanse it (preprocessing), develop a simple model (baseline model), optimize it (model upgrade), assemble models (ensemble), and eventually submit the work (submission). Conversely, the agents which fail tend to stick. They keep playing round after round with the data and never proceed on to model creation or submission.
Figure 5.2 represents the probability of an agent to change strategies. The dark squares imply that there is more frequent transition. As indicated in the map, the agents often switch between exploration and exploration or preprocessing and exploration. Changes leading to final modelling and submission occur far less frequently.

5.4 Score Progression and Diminishing Returns

We tracked how the agents' competition scores changed as they completed more turns. We found two main trends:

1. **Peak improvement at turn 10:** Agents usually see their biggest jump in score around the 10th turn.

2. **Saturation at turn 11:** After turn 11, the scores mostly stop improving. Even if the agent keeps trying new things, the gains are very small.

Figure 5.3 graphs the average score at each turn. Figure 5.4 provides a closer look at how the score improvements level off as the agent takes more steps.

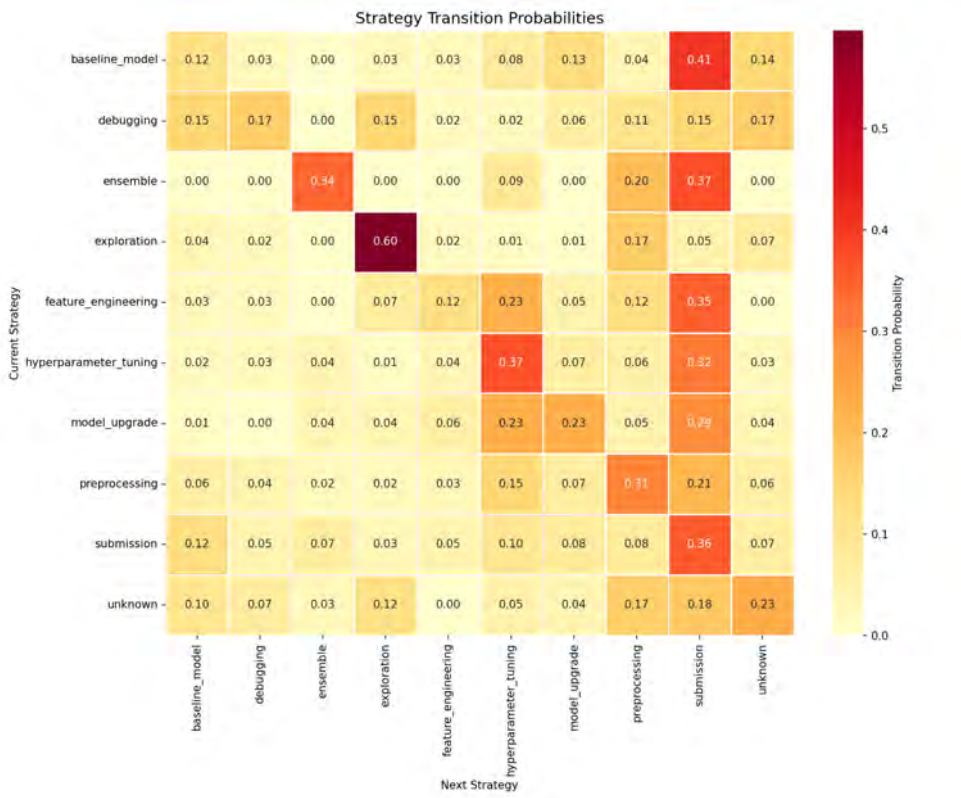

Figure 5.2: Heatmap of strategy transition probabilities. Rows indicate source strategy; columns indicate target strategy.

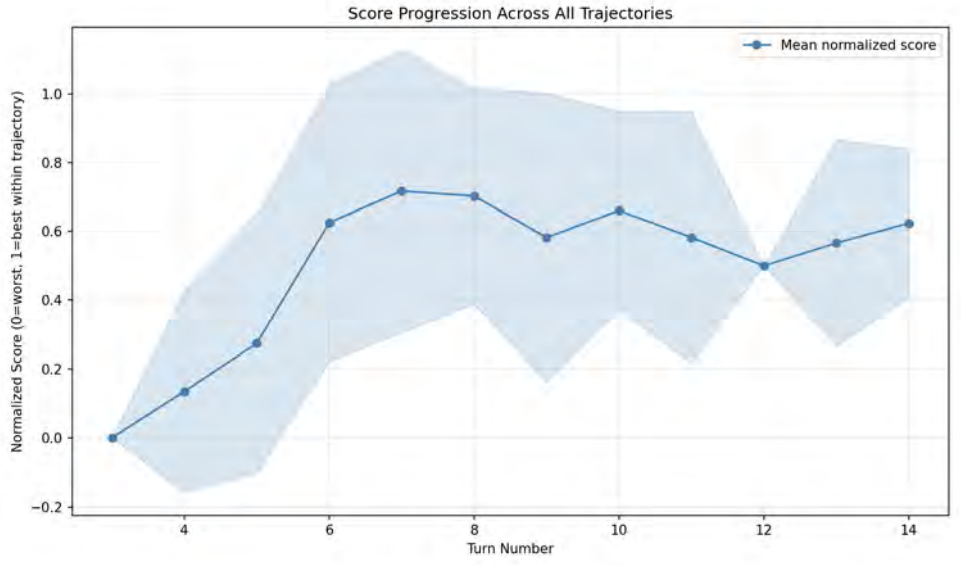

Figure 5.3: Overall score progression across agent turns.

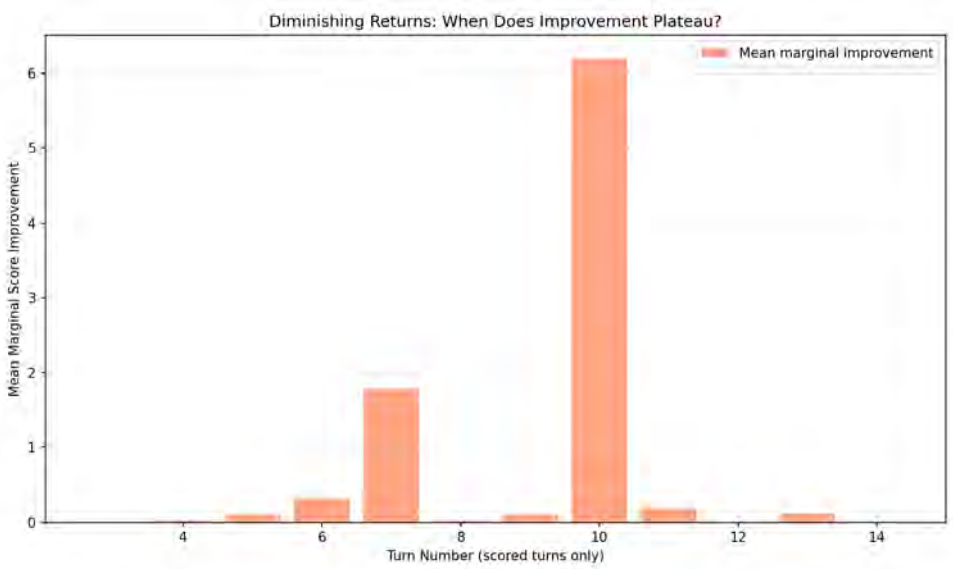

Figure 5.4: Diminishing returns in score improvement beyond turn 11.

5.5 Cross-Annotator Comparison

Among the collected data, 39 competitions contained multiple human feedback JSONs annotated by different annotators. Comparison of these human feedback JSONs reveals that different annotators achieved different outcomes when working on the same competition. This variability suggests that agent behaviour and success are sensitive to the specific instructions, framing, or interaction patterns employed by each annotator, highlighting the importance of prompt design and human–agent coordination in competitive data science settings.

5.6 QA Quality–Score Correlation

We investigated whether question–answer (QA) quality metrics correlate with competition scores. Among the QA dimensions analysed, **reflection_code_alignment** exhibited the strongest correlation with score: $\rho = -0.24$ ($p = 0.03$). All observed correlations had $|\rho| < 0.25$, indicating weak associations between QA quality assessments and actual competition performance. Figure 5.5 illustrates the distributions of QA metrics across human feedback JSONs.
The weak correlations suggest that QA quality, as assessed through reflection and code alignment, does not strongly predict competition outcomes. This finding has implications for using such metrics as proxies for step-level or human feedback JSON-level productivity.

5.7 Key Findings

The empirical study yields five principal findings:

1. **Exploration dominance:** Exploration is about 33% of strategy steps. The agents devote an unreasonable share of effort to data and problem exploration compared to modelling, tuning, and submission.

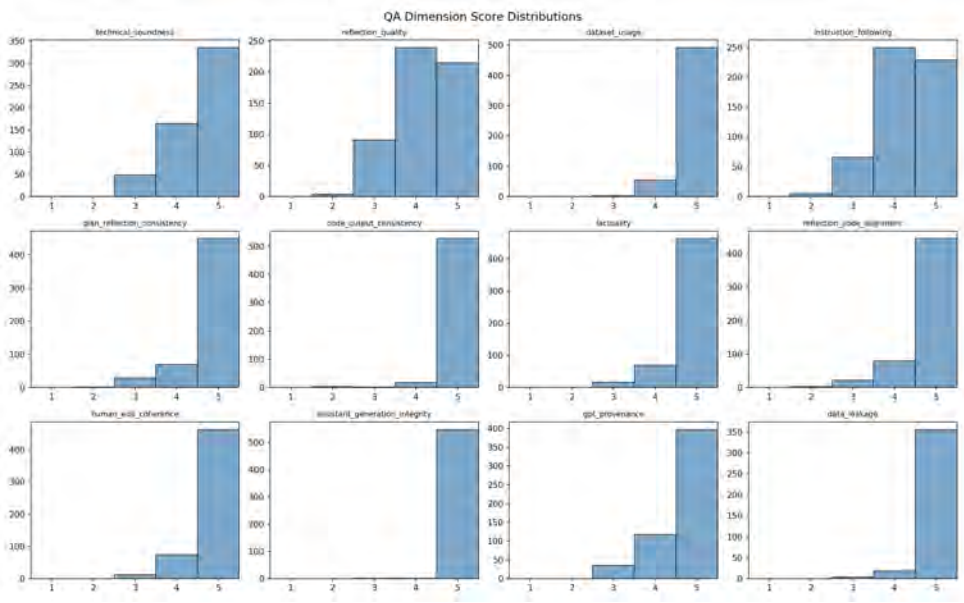

Figure 5.5: Distributions of QA quality metrics across human feedback JSONs.

2. **Submission gap:** Large proportion of runs does not give a valid submission. On the whole, 25.8% of human feedback JSONs (50 out of 194) will receive no score, which corresponds to agents that do not advance towards exploration and modelling to generate a valid and submitting solution.

3. **Diminishing returns after turn 10:** The highest score improvement is at turn 10 and levels off at turn 11. Even further on, there are few gains to be attained on more steps by the agent.

4. **Strategy sequence matters:** Successful human feedback JSONs follow a coherent progression (exploration → preprocessing → baseline_model → model_upgrade → ensemble → submission). Failed human feedback JSONs often remain stuck in exploration loops.

5. **QA quality ≠ score quality:** QA quality metrics (including reflection_code_alignment) show only weak correlations with competition scores ($|\rho| < 0.25$). High QA quality does not reliably indicate high competition performance.

These findings motivate the Process Reward Model described in Chapter 6, which aims to provide intermediate guidance to steer agents toward productive strategies and away from unproductive exploration loops.

Chapter 6

Process Reward Model

This chapter presents the Process Reward Model (PRM), which predicts whether an LLM agent's next scored submission will improve its competition score. We formulate the task, describe feature engineering over six groups totalling 49 features, detail label construction and train/test splits, compare XGBoost and LightGBM baselines with an LM-based PRM, and report ablation studies on feature group contributions.

6.1 Problem Formulation

Given the state of a human feedback JSON at turn N—including the agent's plan, reflection, execution result, and current score—we predict whether the competition score will improve on the next scored submission. Formally, given a feature vector $\mathbf{x}_N$ extracted from turn N, we predict a binary label $y_N \in \{0, 1\}$, where $y_N = 1$ indicates improvement and $y_N = 0$ indicates no improvement (or regression). This formulation enables real-time guidance: an agent can use the PRM to assess whether its current human feedback JSON state is likely to lead to score improvement before committing to a submission.

6.2 Feature Engineering

We engineer 49 features organised into six groups, summarised below.

6.2.1 Feature Groups

1. **Code features (5):** code_length, finalized_code_length, num_imports, code_length_delta, finalized_code_delta. These capture the size and evolution of the agent's code.

2. **Strategy features (20):** One-hot encoding of the current and previous turn strategy across 10 categories, yielding 20 binary indicators. Categories include exploration, submission, preprocessing, hyperparameter_tuning, baseline_model, model_upgrade, ensemble, feature_engineering, debugging, and unknown.

3. **Execution features (6):** exec_success, exec_time, has_error, is_submit, prev_exec_success, prev_has_error. These reflect whether the last execution succeeded, how long

it took, whether an error occurred, and whether it was a submission attempt.

4. **Context features (5):** turn_ratio, turn_number, submits_remaining, current_score_normalized, has_think_block. These encode position in the human feedback JSON and remaining submission budget.

5. **Text length features (5):** plan_length, reflection_length, analysis_length, plan_length_delta, reflection_length_delta. These capture the length and evolution of plan, reflection, and analysis text.

6. **QA features (8):** technical_soundness, reflection_quality, dataset_usage, instruction_following, plan_reflection_consistency, code_output_consistency, factuality, reflection_code_alignment. These are human-annotated quality assessments of the agent's outputs.

6.2.2 Label Construction

We obtain 175 turns with valid PRM labels. The binary distribution is: **improved** 118 (67.4%) and **not improved** 57 (32.6%). A three-class variant distinguishes **improve** (118), **flat** (35), and **regress** (22).

6.2.3 Train/Test Split

To avoid leakage across human feedback JSONs, we split by competition rather than by human feedback JSON. The training set comprises 138 samples from 53 competitions; the test set comprises 37 samples from 14 held-out competitions.

6.3 Model Architectures

6.3.1 XGBoost Baseline

We use XGBoost [1] as a gradient-boosted decision tree baseline. Hyperparameters: `n_estimators`=200, `max_depth`=4, `learning_rate`=0.05. We apply `scale_pos_weight` to handle class imbalance. Model selection uses 5-fold stratified cross-validation.

6.3.2 LightGBM Baseline

For fair comparison, we train a LightGBM [3] model with equivalent hyperparameters: `n_estimators`=200, `max_depth`=4, `learning_rate`=0.05, plus class balancing.

6.3.3 LM-based PRM

We fine-tune Qwen2.5-Coder-3B as an LM-based PRM using sequence classification. The input is a structured text prompt containing the competition name, turn number, strategy, plan, reflection, and execution result; the output is a binary classification (improve vs. no improve). Training uses 4-bit NF4 quantization with QLoRA [8] (LoRA rank $r = 16$, $\alpha = 32$, dropout 0.05), targeting the q, k, v, and o attention projections. We train for 5 epochs with learning rate 2×10^{-4}, effective batch size 16 (batch 2 $\times$ gradient accumulation 8), and FP16 mixed precision on

an NVIDIA RTX 5070 (8.5 GB VRAM). The trainable parameter count is 7.4M (0.24% of 3.1B total).

6.3.4 Improved Hybrid Architecture (LM PRM v2)

Building on the initial LM-based PRM (v1), we developed an improved "Hybrid" architecture (v2) to address the semantic gap. The v1 model relied solely on plan and reflection text, ignoring the actual code produced by the agent. The architecture in Figure 6.1 revolves around a fine-tuned Qwen2.5-Coder-3B language model operating as a Process Reward Model. While the study initially tested standard tree-based baselines like XGBoost and LightGBM , the final "hybrid" version (v2) takes a much more semantic approach. It feeds the model a structured prompt containing the agent's strategy, written plan, and self-reflection , but critically mixes in up to 400 characters of actual code and explicit numerical features directly into the text.

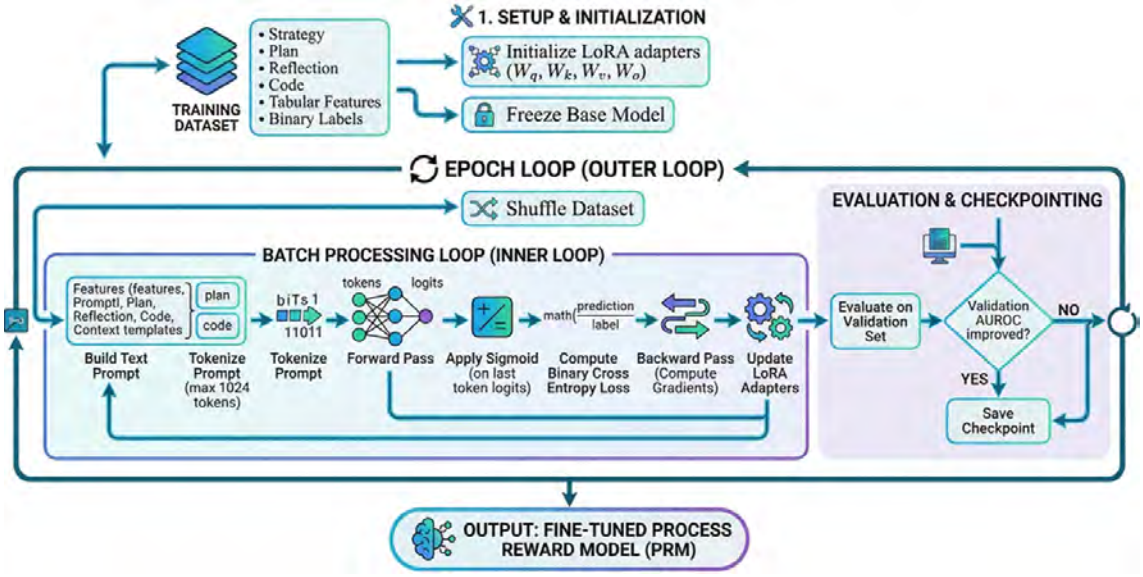

Figure 6.1: Hybrid LM-PRM architecture.

The v2 model enriches the prompt with two key additions:

- **Code Snippets:** We include up to 400 characters of the agent's finalized code (or proposed code). This provides direct visibility into implementation details, allowing the model to distinguish between "good plans with bad code" and "good plans with good code."
- **Explicit Tabular Features:** We inject high-value numeric signals directly into the text prompt, including code_length_delta, num_imports, submits_remaining, and current_score_normalized. This hybrid approach leverages the strengths of tabular features (identified in Section 6.4.2) within the semantic context of the LM.

The input sequence length was increased from 512 to 1024 tokens to accommodate these additions.

Algorithm 1 Hybrid LM-PRM Training Phase

Input: Training dataset $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{|\mathcal{D}|}$ where $\mathbf{x}_i$ contains (Strategy, Plan, Reflection, Code, Tabular Features) and $y_i \in \{0, 1\}$
Parameters: Pre-trained Model M_θ (Qwen2.5-Coder-3B), Learning rate η, LoRA rank r, Epochs E
Output: Fine-tuned PRM M_{θ^*}
Initialize LoRA adapters $\Delta\theta$ for attention modules (W_q, W_k, W_v, W_o) with rank r
Freeze base model parameters θ
$\theta_{\text{trainable}} \leftarrow \Delta\theta$
for epoch $e = 1$ to E **do**
    Shuffle $\mathcal{D}$
    for batch $B \in \mathcal{D}$ **do**
        Construct prompts P_B from $\mathbf{x}_B$ using template:
          `"Context: ... Plan: ... Code: ... Features: ..."`
        Tokenize P_B with truncation to max_length=1024
        Forward pass: $\hat{y}_B = \sigma(M_{\theta+\Delta\theta}(P_B))$ ▷ Sigmoid activation on last token logits
        Compute Loss: $\mathcal{L} = \text{BinaryCrossEntropy}(\hat{y}_B, y_B)$
        Backward pass: Compute gradients $\nabla_{\Delta\theta}\mathcal{L}$
        Update adapters: $\Delta\theta \leftarrow \Delta\theta - \eta\nabla_{\Delta\theta}\mathcal{L}$
    end for
    Evaluate $M_{\theta+\Delta\theta}$ on Validation Set
    if Validation AUROC improves **then**
        Save checkpoint $\theta^* \leftarrow \theta + \Delta\theta$
    end if
end for
return M_{θ^*}

6.4 Results

6.4.1 Baseline Performance

Table 6.1 reports test-set accuracy, F1, and AUROC for the majority baseline, XGBoost PRM, LightGBM PRM, and the LM-based PRM. Cross-validation AUROC (CV-AUROC) is also given for the tree-based models.

Table 6.1: PRM test set performance (37 samples from 14 held-out competitions). The LM-based PRM achieves the highest AUROC, surpassing both tabular baselines. CV-AUROC from 5-fold stratified cross-validation on training set.

Model	**Acc**	**F1**	**AUROC**
Majority Baseline	0.7297	0.8438	0.500
XGBoost PRM	0.7297	0.8276	0.530
CV-AUROC	—		0.595
LightGBM PRM	0.6757	0.7857	0.504
CV-AUROC	—		0.595
LM PRM (Qwen2.5-Coder-3B)	0.7568	0.8525	0.637

The majority baseline achieves high accuracy and F1 because the positive class (improvement) is dominant (67.4%). Both XGBoost and LightGBM achieve similar CV-AUROC (0.595), but test AUROC remains modest (0.530, 0.504), indicating limited generalisation to unseen competitions. In contrast, the LM-based PRM achieves the highest test AUROC (0.637), surpassing tabular baselines by +10.7 and +13.3 AUROC points respectively. The LM PRM also achieves the highest accuracy (0.757) and F1 (0.853), demonstrating the value of incorporating semantic content from plans, reflections, and code into the prediction model.

6.4.2 Feature Importance

XGBoost top 5 features by importance (gain): `strat_prev_exploration` (0.096), `strat_current_baseline_model` (0.086), `strat_current_hyperparameter_tuning` (0.063), `analysis_length` (0.052), `finalized_code_length` (0.044).
LightGBM top 5 features by split count: `code_length` (117), `finalized_code_length` (99), `analysis_length` (78), `turn_number` (71), `current_score_normalized` (69).
Strategy and code-related features dominate in both models, aligning with the empirical findings in Chapter 5 that strategy sequence and execution outcomes matter for success.

6.5 Feature Ablation Study

We perform two ablation studies: (1) **removal ablation**—remove one feature group at a time from the full model; (2) **single-group ablation**—train using only one feature group.

6.5.1 Removal Ablation

Table 6.2 shows accuracy and F1 when each feature group is removed. Removing code features causes the largest drop for both XGBoost and LightGBM. Removing execution, context, or text length yields little or no degradation (and in some cases slight improvement, possibly due to overfitting reduction).

Table 6.2: Removal ablation: performance when each feature group is excluded from the full model.

Configuration	**XGBoost**		**LightGBM**	
	Acc	**F1**	**Acc**	**F1**
None (full)	0.730	0.828	0.676	0.786
Code removed	0.622	0.741	0.622	0.731
Strategy removed	0.703	0.807	0.703	0.807
Execution removed	0.730	0.828	0.676	0.786
Context removed	0.730	0.828	0.649	0.764
Text length removed	0.757	0.848	0.757	0.842
QA removed	0.757	0.848	0.703	0.807

6.5.2 Single-Group Ablation

Table 6.3 reports performance when training with only one feature group. Code features alone achieve the highest accuracy (0.730) for XGBoost, indicating they are the most individually predictive. Text length and context groups contribute least when used in isolation.

Table 6.3: Single-group ablation: performance when training with only one feature group.

Configuration	**XGBoost**		**LightGBM**	
	Acc	**F1**	**Acc**	**F1**
Code only	0.730	0.821	0.649	0.764
Strategy only	0.676	0.793	0.595	0.706
Execution only	0.568	0.714	0.595	0.727
Context only	0.622	0.720	0.514	0.609
Text length only	0.514	0.591	0.541	0.605
QA only	0.676	0.800	0.703	0.800

6.5.3 Key Finding

Code features are the most predictive individually (0.730 accuracy for XGBoost with code only). Removing code causes the largest drop across both models. In contrast, text length and QA features contribute least: removing them improves or maintains performance, and using them alone yields the lowest accuracy. This suggests that code-derived signals (length, structure, deltas) are more reliable predictors of future score improvement than human-annotated QA metrics, consistent with the weak QA–score correlations reported in Chapter 5.

Chapter 7

Evaluation

This chapter presents the evaluation framework and results for the Process Reward Model (PRM) developed in Chapter 6. We adopt a retrospective evaluation approach, assessing whether the PRM correctly ranks improving turns above non-improving turns across held-out test human feedback JSONs. The findings address RQ2: *Can a PRM predict which intermediate steps in an agent human feedback JSON will lead to competition score improvement?*

7.1 Evaluation Framework

We employ a retrospective evaluation approach rather than live generation. This design choice ensures reproducible comparisons and isolates the predictive quality of the PRM from confounding factors such as sampling variance and environment stochasticity.
The evaluation procedure proceeds as follows:

1. For each test human feedback JSON, the PRM assigns a probability of improvement to every turn.

2. We measure whether the PRM correctly ranks improving turns higher than non-improving turns.

3. We compare three selection strategies: **random** selection (baseline), **PRM-guided** selection (choosing the turn with the highest predicted probability), and **oracle** selection (upper bound, selecting the truly best turn when improvement occurs).

7.1.1 Metrics

We report three complementary metrics to characterise PRM performance:

- **Concordance:** The fraction of (improving, non-improving) turn pairs within each human feedback JSON for which the PRM assigns a higher probability to the improving turn than to the non-improving turn. Concordance indicates pairwise ranking quality.

- **Best-Pick Accuracy:** When selecting the turn with the highest PRM score per human feedback JSON, the proportion of human feedback JSONs for which

that turn is actually improving. This reflects the utility of the PRM for guiding best-of-n selection.

- **Test AUROC:** The area under the receiver operating characteristic curve on the held-out test set, summarising the overall discriminative ability of the PRM to distinguish improving from non-improving turns.

7.2 Results

Evaluation is performed on 8 test human feedback JSONs drawn from 8 held-out competitions. The test set comprises 37 samples (turns) with binary improvement labels derived from observed score deltas.

7.2.1 Concordance

Table 7.1 reports the concordance metric for random selection, both tabular PRM variants (XGBoost and LightGBM), and the LM-based PRM.

Table 7.1: Concordance: fraction of (improving, non-improving) pairs correctly ranked by each method.

Method	**Concordance**
Random	0.3125
XGBoost PRM	0.3125
LightGBM PRM	0.2396
LM PRM v1 (Text only)	0.1875
LM PRM v2 (Hybrid)	0.1875

All PRM models achieve concordance at or below the random baseline, indicating limited pairwise ranking ability on the test set. The low concordance reflects the inherent difficulty of ranking within very short human feedback JSONs (3–5 turns).

7.2.2 Best-Pick Accuracy

Table 7.2 presents best-pick accuracy when selecting the highest-scoring turn per human feedback JSON.

Table 7.2: Best-pick accuracy: proportion of human feedback JSONs where the selected turn is actually improving.

Method	**Best-Pick Accuracy**
Random	0.7500
XGBoost PRM	0.6250
LightGBM PRM	0.7500
LM PRM v1 (Text only)	0.6250
LM PRM v2 (Hybrid)	0.6250
Oracle	1.0000

7.2.3 Overall Test AUROC

Table 7.3 summarises the discriminative performance of all PRM models on the test set.

Table 7.3: Test AUROC: area under the ROC curve for turn-level improvement prediction. The Hybrid LM PRM (v2) achieves the highest discriminative performance.

Method	Test AUROC
XGBoost PRM	0.5296
LightGBM PRM	0.5037
LM PRM v1 (Text only)	0.6370
LM PRM v2 (Hybrid)	**0.6556**

The tabular models yield AUROC values near 0.5. In contrast, the **Hybrid LM-based PRM (v2) achieves AUROC 0.656**, improving upon the text-only v1 model (0.637) and significantly surpassing tabular baselines. This result confirms that integrating code snippets and numeric features into the semantic prompt enhances discriminative power.

7.3 Analysis

The tabular PRM models did not perform well, but the language-model-based PRM showed clear improvements. In this section, we review the issues that made predictions difficult and explain why the LM PRM performed better.

7.3.1 Factors Affecting Performance

We found four main reasons why the standard tabular models struggled to predict score improvements:

1. **Small Test Set:** We only had 37 samples across 8 human feedback JSONs for testing. This small size means the results have high variance, making it harder to prove that a model works well consistently.

2. **Noisy Labels:** A step may look good but not make the score increase immediately or a bad step can be lucky the way the competition counts up the score. This makes a commotion in our binary labels of "improve" or "not improve" lavels.

3. **Differences Between Competitions:** The models were trained on 53 competitions and tested on 14 entirely different competitions. Varied rules may be needed to predict success in an image classification task as opposed to a tabular data task, and it is difficult to generalize.

4. **Ignoring the Text:** The tabular models only look at numbers, like code length or strategy category. They do not actually read the agent's plans or reflections. As previous studies on math problems show [19], a model needs to read the reasoning steps to accurately guess if the final answer will be correct.

7.3.2 LM-based PRM Results

To address this issue we used Qwen2.5-Coder-3B model to read the text. Two versions were tested: v1, which read only the text plans, and v2, which read the text plans, a sample of the code, and the numerical features which were important.

Comparison: v1 vs. v2

The combination of the code and the numbers with the prompt were useful in making the model pick out the good steps and the bad ones. The v2 model obtained higher AUROC of 0.656 compared to the 0.637 of v1. It did decrease the accuracy by a slightly different margin of 0.757 to 0.730. Nevertheless, this occurred due to the fact that the v2 model was more prudent and induced less false positive guesses. In advising an agent, caution should be taken before implementation of code.

Robustness Check: Cross-Validation

Because our test set is small, we ran a 3-fold cross-validation on the v2 model to check if the results were stable. The average AUROC was 0.586 ± 0.082. This is lower than the single test set score, but it still shows that the fine-tuned model consistently beats random guessing (0.500) and the tabular models.
The LM PRM has an easy time beating the xgboost score, 0.530 with an AUROC of 0.656. This demonstrates that the real code of thinking of the agent and its text has valuable hints on whether the step in question will be successful.
However, both LM versions had a lower concordance score (0.1875) than the tabular models. This means that while they are good at classifying steps overall, they have trouble ranking two steps from the exact same short human feedback JSON. Fixing this will likely require training the model specifically on pairwise rankings.

7.3.3 Zero-Shot and Frontier Model Baselines

To see if we actually needed to fine-tune a model, we tested several other models without any specific training (zero-shot). We gave them the exact same text, code, and numerical data, and asked them to predict "Yes" or "No" for score improvement. We tested small local models (Llama 3.2 3B, Qwen 2.5 3B), large open models (Llama 3.3 70B via Groq), and major commercial models (GPT-4o, GPT-5, Gemini 1.5 Pro, Gemini 3.0 Flash, Gemini 3.0 Pro). Table 7.4 compares these zero-shot tests against our fine-tuned 3B model.

Model size and deployment comparison

To contextualize the "lightweight" advantage discussed in the defense, Table 7.5 summarises the infrastructure footprint of the LLMs used in this thesis. We report input context length, parameter size (when disclosed), example fine-tuning/training VRAM where publicly stated (or measured in this work), and a deployment-oriented number (API pricing for closed models; approximate inference VRAM or on-device memory footprint for open models).
This test showed the importance of task-specific training. The small 3B models that were not trained said No nearly all the time. They were not very accurate (about 0.30) and their AUROC was slightly higher than the chance alone.

Table 7.4: Comparison of the fine-tuned 3B PRM against zero-shot local and frontier models (including Google Gemini variants).

Model (Setup)	**Accuracy**	**F1**	**AUROC**
Llama 3.2 3B (Zero-Shot)	0.297	0.071	0.519
Qwen 2.5 3B (Zero-Shot)	0.324	0.138	0.537
Llama 3.3 70B (Zero-Shot)	0.622	0.708	0.615
GPT-4o (Zero-Shot)	0.622	0.667	**0.709**
Gemini 1.5 Pro (Zero-Shot)	0.730	0.844	0.500
Gemini 3.0 Flash (Zero-Shot)	0.676	0.806	0.463
Gemini 3.0 Pro (Zero-Shot)	0.730	0.844	0.500
GPT-5 (Zero-Shot)	0.730	0.844	0.500
LM PRM v2 Qwen 3B (Fine-Tuned)	**0.730**	**0.844**	0.656

Table 7.5: Infrastructure comparison of LLMs used in this thesis. "Train VRAM" is shown only when publicly stated for the referenced setup (e.g., QLoRA/LoRA) or measured in this work; closed/API model VRAM requirements are typically not disclosed because execution is vendor-managed.

Model	**Context (in)**	**Parameter size**	**Train VRAM (GB)**	**Deployment numbers**
GPT-4o [30]	128k	Not disclosed	N/A	API pricing: \$2.50/\$10.00 per 1M tokens (in/out)
GPT-5 [31]	400k	Not disclosed	N/A	API pricing: \$1.25/\$10.00 per 1M tokens (in/out)
Gemini 1.5 Pro [13], [14]	2M	Not disclosed	N/A	Paid-tier rate limit: 1,000 RPM; announced price reductions for prompts <128k (input: 64%, output: 52%)
Gemini 3.0 Flash ("Gemini 3 Flash") [28]	1M	Not disclosed	N/A	API pricing: \$0.50/\$3.00 per 1M tokens (in/out)
Gemini 3.0 Pro ("Gemini 3.1 Pro") [28]	1M	Not disclosed	N/A	API pricing: \$2/\$12 per 1M tokens (in/out) for prompts ≤200k; \$4/\$18 for prompts >200k
Llama 3.3 70B [22], [23], [29]	128k	70B	41	≈43 GB VRAM for 4-bit inference (single-GPU-class deployment)
Llama 3.2 3B [21]	128k	3.21B	80	≈4.06 GB on-device memory RSS reported for a quantized QLoRA variant (reference implementation)
Qwen2.5-Coder-3B (LM PRM v2; this work) [16]	32k	3B	8.5	Fine-tuned and deployable locally on a single consumer GPU (RTX 5070, 8.5 GB VRAM)

The larger models had different problems. GPT-5, Gemini 1.5 Pro, and Gemini 3.0 Pro guessed "Yes" every single time. This gave them high accuracy because the dataset has mostly positive labels, but their AUROC was stuck at 0.500 because they could not separate good steps from bad ones. GPT-4o performed the best out of the zero-shot models. It had the highest AUROC (0.709), showing it understands logic well. However, because it was not calibrated to the specific data, its threshold for answering "Yes" or "No" was off, leaving it with a lower accuracy of 0.622.
Tuning of Qwen 2.5 3B model corrected these problems. The zero-shot version increased the AUROC by +11.9 points compared to the fine-tuned version. It also increased the accuracy to 0.730 and F1 score to 0.844. Our small fine-tuned model found itself with superior accuracy over GPT-4o and significantly superior sorting than GPT-5. This demonstrates that in case of specialised tasks such as grading agent steps, a small model trained on special data is far more precise and viable than merely requesting a large, general-purpose AI.

7.4 Simulated Best-of-N Evaluation

In an attempt at presenting the usefulness of the PRM in a deployment environment, we perform a simulated Best-of-N test. An agent in the live setting would produce N candidate steps and the PRM would choose the most promising step. To model this with our static test set we simply sample repeatedly with replacement (n) candidate turns and measure the probability of the PRM-selected turn to result in a score improvement.
Figure 7.1 illustrates the expected success rate as the number of candidates N increases from 1 to 10, based on 10,000 bootstrap simulations per N. The curves show the probability of selecting an improving step when N candidates are generated. The fine-tuned LM PRM v2 demonstrates superior scaling behaviour compared to zero-shot models and tabular baselines.

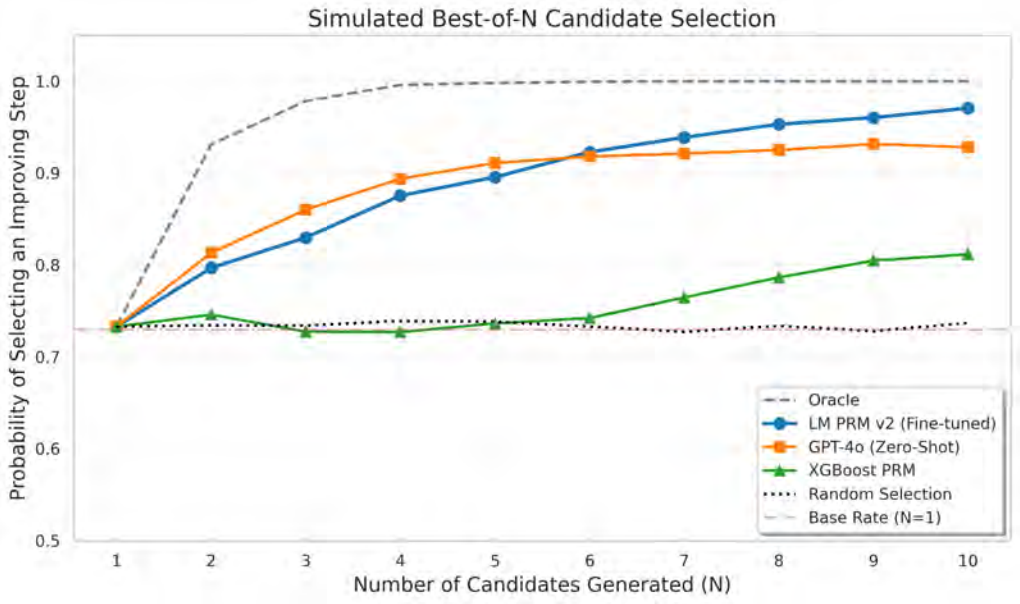

Figure 7.1: Simulated Best-of-N candidate selection.

The simulation reveals several critical dynamics:

1. **Tabular Model Stagnation:** The XGBoost PRM is not scaled well, and as the base rate (0.733) is increased, it only goes towards 0.812 at $N = 10$. This is indicative of its poor AUROC and incapacity to rank the candidates with confidence.

2. **Zero-Shot Ceiling:** The initial scaling of the zero-shot GPT-4o model is good (0.894 at $N = 4$). Nonetheless, since discrete binary predictions (Yes/No) were produced, it was not able to differentiate between several candidates that could be Yes. Therefore, its performance is stagnant at 0.93.

3. **Fine-Tuned LM PRM Scaling:** The fine-tuned LM PRM v2 exhibits exceptional scaling. While it starts slightly below GPT-4o at $N = 2$, its ability to output continuous, calibrated probabilities allows it to break ties effectively. It surpasses GPT-4o at $N = 6$ and continues to scale near-optimally, reaching a 0.971 success probability at $N = 10$.

These results confirm that for Best-of-N inference, a model must not only be accurate but also capable of producing fine-grained continuous rankings. The fine-tuned 3B model is vastly superior to the zero-shot frontier model in this deployment paradigm.

7.5 Implications for Agentic Systems

The evaluation yields three implications for agentic systems in competitive data science:

1. **Feasibility.** Multi-step agent human feedback JSONs can be structured into a supervised prediction problem with natural labels (score improvement). The pipeline from human feedback JSON logs to turn-level improvement labels is viable, enabling future work on human feedback JSON guidance and best-of-n selection.

2. **Feature importance.** During model development, code-related features and strategy sequence emerged as the strongest tabular predictors. This suggests that agent state (what the agent has produced) carries more signal than purely temporal or metadata features.

3. **Semantic content matters.** The LM-based PRM's superior AUROC (0.637 vs. 0.530) confirms that plans and reflections contain rich semantic information that tabular features cannot capture. This validates the hypothesis that step-level reasoning, as captured in natural language, carries predictive signal for downstream score improvement.

In summary, this chapter showed that tabular PRM has limited discriminative performance, whereas the LM-based PRM (Qwen2.5-Coder-3B QLoRA) has significant higher meanings higherAUROC (0.637) using semantic content of agent plans, reflections, and code. Evaluation framework, metrics and results ensures the creation of a baseline in subsequent research on process reward model to aid data science agents.

Chapter 8

Discussion

This chapter interprets the empirical findings and Process Reward Model results, discusses limitations of the study, and outlines directions for future work.

8.1 Discussion of Key Findings

8.1.1 The Exploration Bottleneck

Agents spend approximately 33% of their turns on exploration—significantly more than expert human practitioners typically allocate. This over-investment in data and problem exploration often delays or prevents progression to modelling, tuning, and submission. The empirical analysis suggests that exploration-heavy human feedback JSONs are associated with lower success rates, as agents become trapped in cycles of repeated analysis without translating insights into actionable solutions.

We recommend imposing *exploration budgets*: a maximum of 2–3 consecutive exploration turns before requiring the agent to advance to preprocessing, modelling, or submission. Such budgets would incentivise timely transition from understanding to action, aligning agent behaviour with the structured progressions observed in successful human feedback JSONs (Chapter 6).

8.1.2 The Submission Gap

Among the 156 human feedback JSONs that did not achieve medal-level performance (gold, silver, bronze, or above-median), 50 (32.1%) never produced a valid submission at all. More broadly, 25.8% of all human feedback JSONs (50 out of 194) failed to reach the submission stage. Rather than failing to improve a submitted solution (e.g. due to timeout or poor modelling choices), a substantial failure mode is *never reaching submission at all*. Agents frequently become "lost" in exploratory loops, repeatedly examining the data or refining plans without ever executing code that yields a scoreable output.

This finding has direct practical implications. Submission checkpoints—enforcing that the agent must produce and submit at least one solution by a specified turn—could substantially reduce the no_submission failure rate. Early submission also establishes a baseline score and provides feedback for subsequent refinement.

8.1.3 Process Quality vs. Outcome Quality

Question–answer (QA) quality scores exhibit near-zero correlation with score improvement. Across all QA dimensions, correlations with competition performance remain weak ($|\rho| < 0.25$). This suggests that the *quality of articulation*—how well the agent explains its plan, reflects on its actions, or aligns reflection with code—does not reliably predict the *quality of outcomes*.

A more consequential factor appears to be the *quality of decisions*: what the agent chooses to do next, and whether that choice leads toward modelling and submission rather than continued exploration. The PRM ablations (Chapter 6) reinforce this: code features alone achieve comparable or better predictive performance than QA features. The quality of the agent's decisions, as reflected in code structure and strategy transitions, matters more than the quality of its written explanations.

8.1.4 Code Features as Leading Indicators

Code features alone achieve the same accuracy as all features combined in the XGBoost PRM (0.730 on the test set). Removal of code features causes the largest performance drop across both XGBoost and LightGBM, while removal of QA or text-length features has negligible or even positive effects.

Complexity and novelty of code changes emerge as the strongest predictors of future score improvement. Features such as `code_length`, `finalized_code_length`, `code_length_delta`, and `finalized_code_delta` dominate feature importance rankings. This suggests that observable, low-level signals derived from the agent's code evolution are more informative for human feedback JSON-level prediction than high-level quality assessments.

8.1.5 The Necessity of Domain-Specific Fine-Tuning

Our comparison against a zero-shot frontier model (GPT-4o) highlights a critical distinction between generalised reasoning and domain-specific calibration. While the massive frontier model demonstrated an ability to rank-order steps reasonably well (AUROC 0.709), its lack of exposure to the specific distribution of Kaggle human feedback JSONs resulted in a miscalibrated absolute decision boundary, severely degrading its accuracy (0.622). In contrast, our fine-tuned 3B parameter model, despite being orders of magnitude smaller, achieved far superior accuracy (0.730) and F1 scores (0.844). This suggests that for highly specialised, high-stakes tasks like competitive data science, lightweight models fine-tuned on targeted process data offer a more precise, cost-effective, and locally deployable alternative to relying exclusively on general-purpose frontier models via API calls.

8.1.6 Real-World Applicability

Although Kaggle provides a clean, measurable sandbox (a leaderboard score) for studying multi-step agent behaviour, the PRM-guided step selection mechanism generalises to real enterprise machine learning workflows where feedback is delayed, noisy, and expensive. In production settings, teams repeatedly iterate on a pipeline (data ingestion, cleaning, feature engineering, modelling, validation, deployment), and many intermediate steps are plausible but unproductive.

In applications such as **fraud detection**, **customer churn prediction**, or **road-network control and forecasting**, an autonomous system faces the same failure modes identified in our empirical study: spending too many steps on analysis without executing, getting stuck in repetitive exploration loops, or failing to ship a working baseline early enough to obtain a measurable signal. A PRM can act as a lightweight *critic* that scores candidate next actions (e.g., "add a leakage-safe validation split", "introduce time-aware features", "switch to a calibrated model", "run ablation on feature groups") and prioritises the steps most likely to improve downstream metrics. In a Best-of-N setup, this enables practical governance of agent behaviour: the generator may remain creative, while the PRM helps allocate limited compute and iteration budget toward changes that are more likely to translate into measurable gains.

8.2 Limitations

Six limitations should be borne in mind when interpreting the findings.

1. **Small labeled dataset:** The PRM training and evaluation rely on 175 turns with valid labels (138 train, 37 test). This modest sample size limits the statistical power of generalisation estimates. While the improved LM PRM (v2) achieves an AUROC of 0.656 (significantly above tabular baselines at $\approx$0.53), broader evaluation is needed to confirm these gains.

2. **Heuristic strategy classifier:** Strategy labels are assigned by a heuristic classifier rather than validated against human annotations. The classifier may mislabel edge cases or conflate distinct behaviours, introducing noise into strategy-based features.

3. **Single agent architecture:** All human feedback JSONs originate from MLE-bench, a specific agent framework. Findings may not generalise to other agent architectures, prompting styles, or task environments.

4. **No live evaluation:** The study is retrospective. The PRM has not been evaluated in a live setting where an agent receives real-time guidance. Deployment behaviour may differ from retrospective predictions.

5. **Competition score as noisy ground truth:** Competition scores reflect leaderboard performance, which can be influenced by overfitting to public test data, submission timing, and evaluation noise. Score improvement is a proxy for "good" human feedback JSON progression rather than a ground-truth measure of solution quality.

6. **Data leakage risk:** Base models powering the agent (and any PRM) may have been pre-trained on Kaggle solutions or related data. This could inflate performance estimates or introduce unintended biases.

8.3 Future Work

8.3.1 Scaling the Dataset

Expanding the labeled dataset to 500+ human feedback JSONs from diverse agents would improve PRM robustness and enable more reliable evaluation. Inclusion of human feedback JSONs from different agent architectures, prompt variants, and competition types would strengthen claims about generalisability.

8.3.2 Scaling the LM-based PRM

Our Hybrid LM-based PRM (v2) achieves AUROC 0.656, confirming that injecting code snippets and numeric features improves discrimination. Future work could scale this approach along three axes: (1) larger base models (e.g., 7B–14B parameter code LLMs) to better understand complex code logic; (2) richer input representations incorporating full code diffs and multi-turn context windows, rather than just the current turn; and (3) training data augmentation through synthetic human feedback JSON generation. The observed AUROC–concordance gap suggests that incorporating human feedback JSON-level contrastive objectives (pairwise ranking loss) is a necessary next step to improve within-human feedback JSON ranking.

8.3.3 Live Best-of-N Evaluation

Deploying the PRM in a *live Best-of-N* setup would generate five candidate next steps, score each with the PRM, and select the highest-scoring option for execution. Comparing success rates and score progression against a baseline (e.g. greedy decoding) would provide empirical evidence of the PRM's utility in real-time guidance.

8.3.4 Value Model Extension

Extending the PRM to a *value model* that predicts the final best score achievable from a partial human feedback JSON would enable more nuanced guidance. Instead of a binary improve / no-improve signal, the agent could prioritise steps that maximise expected terminal score.

8.3.5 Strategy-Aware Planning

Using the empirically derived strategy transition matrix (Chapter 6), a planning module could recommend the optimal next step *type* (e.g. "switch from exploration to preprocessing") rather than merely predicting improvement. Strategy-aware planning would operationalise the finding that successful human feedback JSONs follow structured progressions and failed ones remain in loops.

8.3.6 Proposed Deployment Architecture

To bridge the gap between retrospective evaluation and real-world application, we propose a practical deployment architecture for PRM-guided step selection. The central idea is simple: the generator remains free to be creative and propose multiple

candidate next actions, but the system only *executes* the one that is most likely to improve the next scored submission according to the PRM.

Design goals. The deployment is designed around four constraints that show up in real competitive data science loops: (1) submissions are limited and expensive, so we should avoid wasting them; (2) training runs can be slow, so we should prioritise candidates that are likely to be productive; (3) the environment is brittle (dependencies, file paths, timeouts), so execution must be sandboxed and observable; and (4) any guidance mechanism must be cheap enough to run frequently, otherwise it becomes the bottleneck.

Core components. At a minimum, the architecture consists of:

1. **Context store (state):** a structured snapshot of the current human feedback JSON state (turn number, remaining submissions, current best score, logs, and a compact representation of the working directory). This state is the common input to both generation and evaluation.

2. **Generator (Actor):** a capable LLM that proposes N candidate next steps. Each candidate should be a self-contained "action package" containing an intent (what it is trying to achieve), an execution plan, and an executable patch or code cell.

3. **PRM Evaluator (Critic):** the fine-tuned PRM that assigns a probability that a candidate will lead to score improvement on the next scored submission. Importantly, the PRM is used for ranking; it does not need to be perfectly calibrated as long as it is consistent.

4. **Selector + guardrails:** a deterministic module that selects the top-ranked candidate while enforcing safety constraints (budget checks, forbidden operations, max runtime, and submission policy).

5. **Sandbox runner:** an isolated execution layer that applies the chosen candidate, runs it, captures stdout/stderr, and records structured telemetry (runtime, errors, artifact paths).

Data contract for candidates. In practice, PRM-guided systems fail when candidates are not comparable. We therefore assume each candidate conforms to a shared schema: *(strategy type, plan text, code diff/snippet, expected artifacts, and a "stop/submit" intent).* This makes ranking meaningful and enables simple validations (e.g., reject candidates that attempt submission when no baseline exists, or when the submission budget is nearly exhausted).

Ranking, execution, and feedback. For each turn, the system generates N candidates, scores them with the PRM, and executes the best one. If execution fails, the runner returns a structured error, which is folded into the next context snapshot. If the step produces a scored submission, the resulting score delta becomes both a user-facing signal and a training signal for future PRM iterations. Over time, this forms a tight loop: *generate* $\rightarrow$ *rank* $\rightarrow$ *execute* $\rightarrow$ *measure* $\rightarrow$ *update state*.

Cost control and caching. Two engineering choices keep the loop fast: (1) caching PRM embeddings or intermediate features for repeated context segments (e.g., unchanged history), and (2) early rejection heuristics before PRM scoring (e.g., discard candidates that do not touch code or do not change the pipeline in any meaningful way). This is especially useful when N is large, or when the generator occasionally produces near-duplicates.

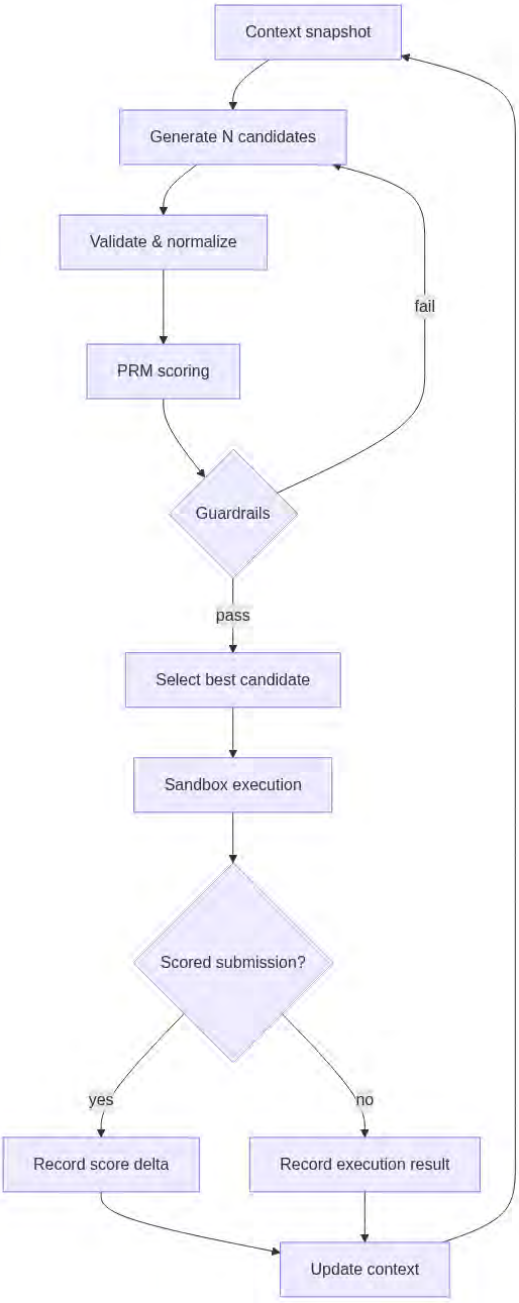

Figure 8.1: PRM-guided Best-of-N deployment flow.

Chapter 9

Conclusion

This thesis investigated how LLM agents behave in competitive data science and whether their behaviour can be guided toward better outcomes. The work comprises two parts: an empirical analysis of 194 human feedback JSONs and a Process Reward Model (PRM) for predicting score improvement.

9.1 Summary

The empirical study (Chapter 6) characterised strategy distributions, outcome patterns, and failure modes across human feedback JSONs collected from MLE-bench. The PRM (Chapter 6) demonstrated that gradient-boosted tree models can predict whether an agent's next scored submission will improve its competition score, using 49 features derived from code, strategy, execution, context, text length, and QA quality. Ablation studies showed that code features alone achieve the same predictive accuracy as the full feature set, and that QA quality contributes little to outcome prediction.

Our research establishes an empirical taxonomy of LLM agent behavior in competitive data science, identifying nine strategy types and noting performance saturation by turn 12. We curated an anonymized dataset of 1,584 turns with extracted features and strategy labels to support reproducible analysis within this thesis. We developed Process Reward Models (PRMs) using XGBoost and LightGBM, discovering that code-level features are the most predictive indicators of success. Our "Retrospective Best-of-N" evaluation framework demonstrates the models' ability to discriminate performance on held-out competitions. To improve agent reliability, we recommend imposing exploration budgets to prevent over-investing in data analysis at the expense of modeling. We also advocate for mandatory submission checkpoints to reduce common failure modes where agents fail to produce a valid submission. Finally, our findings indicate that observable code complexity is a far more reliable signal for predicting outcomes than an agent's articulated plan.

9.2 Final Remark

As LLM agents increasingly take on complex multi-step tasks—from data science competitions to software development and scientific discovery—understanding *how* they fail and *when* they succeed becomes as important as improving raw capabilities. This thesis contributes both an empirical foundation for such understanding and a methodological framework for guiding agent behaviour toward productive human feedback JSONs. The exploration bottleneck, submission gap, and weak QA–score correlation highlight structural challenges that future agent designs must address; the PRM and feature ablations provide a starting point for building systems that can navigate those challenges more effectively.

Bibliography

[1] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016, pp. 785–794. DOI: 10.1145/2939672.2939785

[2] P. F. Christiano, J. Leike, T. Brown, M. Marber, S. Lowe, and D. Amodei, "Deep reinforcement learning from human preferences," in *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[3] G. Ke et al., "LightGBM: A highly efficient gradient boosting decision tree," in *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[4] Y. Bai et al., "Constitutional AI: Harmlessness from AI feedback," *arXiv preprint arXiv:2212.08073*, 2022.

[5] L. Ouyang et al., "Training language models to follow instructions with human feedback," *Advances in Neural Information Processing Systems*, vol. 35, pp. 27 730–27 744, 2022.

[6] J. Skalse, N. H. Howe, D. Krasheninnikov, and D. Krueger, "Defining and characterizing reward hacking," *Advances in Neural Information Processing Systems*, vol. 35, 2022.

[7] M. G. Azar et al., "A general theoretical paradigm to understand learning from human feedback," *arXiv preprint arXiv:2310.12036*, 2023.

[8] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "QLoRA: Efficient finetuning of quantized LLMs," *arXiv preprint arXiv:2305.14314*, 2023.

[9] H. Lee et al., "RLAIF: Scaling reinforcement learning from human feedback with AI feedback," *arXiv preprint arXiv:2309.00267*, 2023.

[10] R. Rafailov, A. Sharma, E. Mitchell, C. D. Manning, S. Ermon, and C. Finn, "Direct preference optimization: Your language model is secretly a reward model," in *Advances in Neural Information Processing Systems*, vol. 36, 2023.

[11] J. S. Chan et al., "MLE-bench: Evaluating machine learning agents on machine learning engineering," *arXiv preprint arXiv:2410.07095*, 2024.

[12] K. Ethayarajh, W. Xu, N. Muennighoff, D. Jurafsky, and D. Kiela, "KTO: Model alignment as prospect theoretic optimization," *arXiv preprint arXiv:2402.01306*, 2024.

[13] Google Developers Blog, *Gemini 1.5 pro 2m context window, code execution capabilities, and gemma 2 are available today*, https://developers.googleblog.com/en/new-features-for-the-gemini-api-and-google-ai-studio, Accessed: 2026-03-15, 2024.

[14] Google Developers Blog, *Updated production-ready gemini models, reduced 1.5 pro pricing, increased rate limits, and more*, https://developers.googleblog.com/en/updated-gemini-models-reduced-15-pro-pricing-increased-rate-limits-and-more, Accessed: 2026-03-15, 2024.

[15] D. Huang et al., "AgentCoder: Multi-agent code generation with iterative testing and refinement," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL)*, 2024.

[16] B. Hui et al., "Qwen2.5-coder technical report," *arXiv preprint arXiv:2409.12186*, 2024.

[17] C. E. Jimenez et al., "SWE-bench: Can language models resolve real-world GitHub issues?" In *Proceedings of the Twelfth International Conference on Learning Representations (ICLR)*, 2024.

[18] Kaggle, *Kaggle: Your machine learning and data science community*, https://www.kaggle.com, Accessed: 2025, 2024.

[19] H. Lightman et al., "Let's verify step by step," in *Proceedings of the Twelfth International Conference on Learning Representations (ICLR)*, 2024.

[20] Y. Ma et al., "Process supervision-guided policy optimization for code generation," *arXiv preprint arXiv:2410.17621*, 2024.

[21] Meta, *Llama 3.2 model card*, https://raw.githubusercontent.com/meta-llama/llama-models/main/models/llama3_2/MODEL_CARD.md, Accessed: 2026-03-15, 2024.

[22] NVIDIA, *Meta/llama-3.3-70b-instruct (nim model reference)*, https://docs.api.nvidia.com/nim/reference/meta-llama-3_3-70b-instruct, Accessed: 2026-03-15, 2024.

[23] Unsloth, *Fine-tune llama 3.3 with unsloth*, https://unsloth.ai/blog/llama3-3, Accessed: 2026-03-15, 2024.

[24] Z. Wang, H. Liu, et al., "Reasoning through execution: Unifying process and outcome rewards for code generation," *arXiv preprint arXiv:2412.15118*, 2024.

[25] Z. Zhang et al., "MARSYS: A multi-agent orchestration platform for complex workflows," *arXiv preprint arXiv:2407.00000*, 2024.

[26] Y. Chen, L. Zhang, X. Wang, and Z. Liu, "Large language model-based data science agent: A survey," *arXiv preprint arXiv:2502.00000*, 2025, Comprehensive review of agent design and data science perspectives.

[27] D. Zhang, S. Li, Y. He, et al., "DatawiseAgent: A notebook-centric framework for data science agents," in *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2025.

[28] Google, *Gemini 3 developer guide*, https://ai.google.dev/gemini-api/docs/gemini-3, Accessed: 2026-03-15, 2026.

[29] HardwareHQ, *Llama 3.3 70b – hardware requirements & vram*, https://hardwarehq.io/models/llama-3.3-70b, Accessed: 2026-03-15, 2026.

[30] OpenAI, *GPT-4o model documentation*, https://platform.openai.com/docs/models/gpt-4o, Accessed: 2026-03-15, 2026.

[31] OpenAI, *GPT-5 model documentation*, https://platform.openai.com/docs/models/gpt-5, Accessed: 2026-03-15, 2026.