MLE-WORLD: PREDICTING MACHINE LEARNING EXPERIMENT OUTCOMES WITH A LEARNED WORLD MODEL

Anonymous authors
Paper under double-blind review

ABSTRACT

AI research agents can propose, execute, and improve solutions to machine learning tasks, but they face a critical bottleneck: execution cost. An agent must wait for each candidate's execution to finish before it receives feedback, which makes it hard to scale at test time by evaluating more candidates. We introduce MLE-World, a learned world model for machine learning engineering that predicts a candidate solution's terminal output (including error traces, training logs, and validation scores) from its code and a description of the task's data, without executing the program. We train MLE-World by supervised fine-tuning Qwen3.5-4B on agent trajectories that pair solution code with observed execution outputs. Evaluating a candidate then requires a single inference call to a 4B model, so an agent can screen several candidates per step and execute only one. Because these predictions are not always accurate, MLE-World also reports a confidence score for each predicted outcome, which we utilize in two ways. First, we propose a confidence gate to only execute solutions with low confidence while relying on the world model to simulate executions with high confidence. Second, because greedily selecting the candidate with the best validation score does not encourage exploration, we propose a selection rule inspired by upper confidence bounds (UCB) to add an exploration bonus based on confidence to predicted validation scores. On 17 known MLE-bench Lite tasks, all five gated policies raise mean leaderboard percentile from 36.9 to 46.4∼49.2, and the gate cuts real execution by 24∼46%.

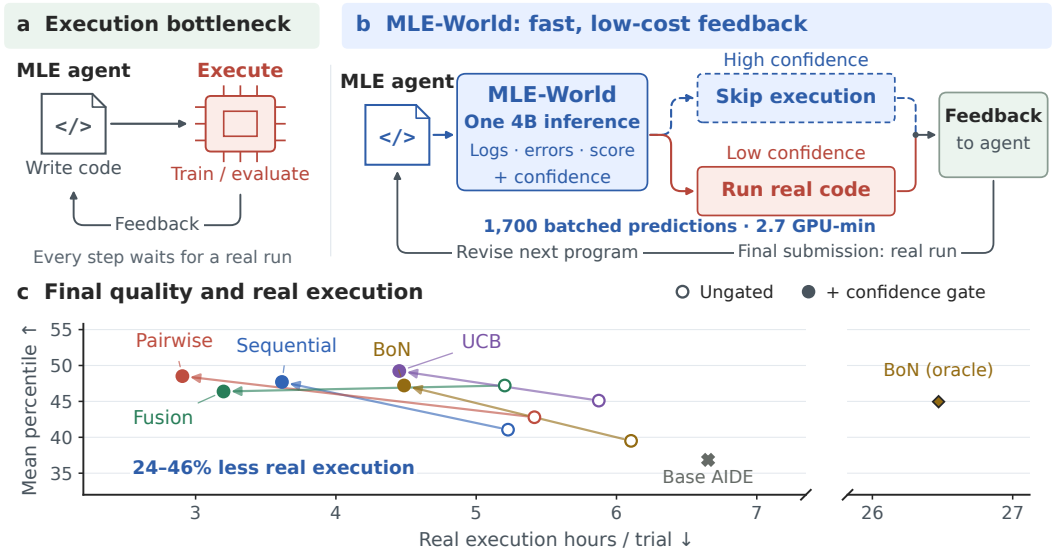


Figure 1: **Fast, cheap feedback with MLE-World.** (a) Execution delays feedback. (b) Predict records and confidence; verify final submissions. (c) Means over 17 tasks × three seeds. Arrows: ungated (open) to gated (filled). All policies use an eight-hour cap except the diamond, a real BoN oracle with a 16-hour cap (Appendix O). Cost sums concurrent runs and cleanup, excluding inference. Broken x-axis; zoomed y-axis.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

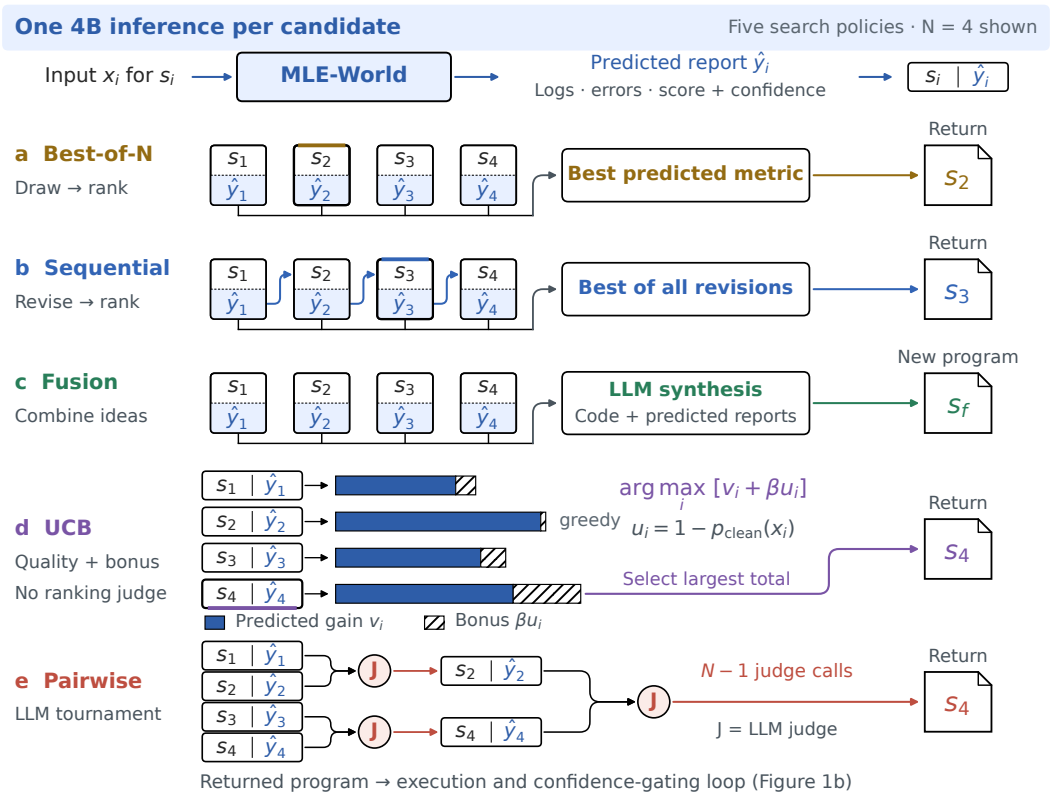


Figure 2: **Five search policies using MLE-World predictions.** Tiles pair code s_i and predicted report $\hat{y}_i$; x_i is the full input (Equation 1). UCB ranks by $v_i + \beta u_i$, with $v_i = v(s_i)$ and $u_i = u(x_i)$. Candidates, bars, and winners are illustrative ($N = 4$, usable predictions). Synthesis and pairwise judging add calls; each policy returns one program to Figure 1(b)'s execution and feedback loop.

1 Introduction

AI research agents repeatedly write code, execute it, inspect terminal output, and revise solutions (Jiang et al., 2025; Nam et al., 2026). Feedback is expensive: each step trains and evaluates a model, yet agents often fail to reach a valid submission (Chan et al., 2025). Execution time limits how many ideas they can test (Toledo et al., 2026; Hambardzumyan et al., 2026; Yang et al., 2026a), making cheap feedback and selective execution central to efficient search.

Prior work prunes unpromising branches (Liu et al., 2025; Toledo et al., 2026), stops training runs early (Kulibaba et al., 2025), or ranks candidates before execution (Zheng et al., 2026; Foster et al., 2026). These methods reduce execution, but a pruned candidate yields no error trace or training log to guide a repair. A world model can predict this feedback without running the program, making machine learning engineering (MLE), with its expensive training runs, a useful setting for such models (Yang, 2026).

MLE-World predicts a candidate solution's terminal output from its code and the task's data description. We fine-tune Qwen3.5-4B on logged code–output pairs from successful and failed runs. One inference call predicts error traces, training logs, and the validation score, supporting both debugging and candidate selection. An agent can thus screen several candidates per step and execute only one.

Prediction errors can mislead search (Janner et al., 2019). MLE-World therefore reports a confidence score: an execution gate trusts high-confidence predictions and requests real feedback otherwise. Confidence also guides selection. Greedy pursuit of validation scores can overfit (Toledo et al., 2026; Hambardzumyan et al., 2026) and overlook candidates the model undervalues. A rule inspired by upper confidence bounds (UCB) adds an uncertainty bonus to predicted quality to encourage exploration.

We evaluate both the world model on its own and the search it guides. On 17 MLE-bench Lite tasks (Chan et al., 2025), MLE-World matches the observed terminal output on 77.9% of held-out programs, against 41.1% for the best prompted baseline, with 13∼963× less GPU time. Inside AIDE, every guided policy submits valid solutions more often than Base AIDE (88∼96% versus 78%). With the confidence gate, all five raise mean leaderboard percentile by 9.5∼12.4 points (2.1∼4.0 SEs, paired by task; four of five survive a Bonferroni correction); without it, only Fusion gains more than 2 SEs. Confidence gating uses 24∼46% less real execution than ungated search.

CONTRIBUTIONS

- **MLE-World**, a 4B world model that predicts a program's terminal output, exception class, and validation score without running it. It matches the real terminal output on 77.9% of held-out programs, against 41.1% for the best prompted baseline, with 13∼963× less GPU time.
- **A confidence gate** that prioritizes real feedback for uncertain predictions. It cuts real execution by 24∼46% against ungated search.
- **A UCB style selection rule** that adds a bonus for outcome uncertainty to predicted quality without a comparative judge. It is competitive with the best policy, and a controlled comparison shows the uncertainty bonus improves over greedy selection (Appendix K).

2 RELATED WORK

MLE agents. MLE agents built on AIDE (Jiang et al., 2025; Nam et al., 2026; Ou et al., 2025) cut execution cost by pruning branches (Liu et al., 2025; Toledo et al., 2026), extrapolating partial training runs (Kulibaba et al., 2025), or ranking candidates before execution (Foster et al., 2026; White et al., 2021). Closest to our search, ForeAgent (Zheng et al., 2026) keeps one of several Improve candidates by pairwise preference with a confidence gate and executes it. A score or preference carries no error trace or training log to guide a repair, and the chosen program still runs; MLE-World predicts that record, and its confidence decides whether a program must run.

World models for LLM agents. World models predict an action's outcome without performing it (Ha & Schmidhuber, 2018; Hafner et al., 2019; Janner et al., 2019). LLM world models, prompted (Thimmaiah et al., 2025; Wang et al., 2024; Gu et al., 2024) or trained on agent logs (Chae et al., 2025; Fang et al., 2025; Yu et al., 2025; Guan et al., 2026; Sun et al., 2026; Zuo et al., 2026), serve web, computer-use, and software agents. Closest to ours, DSWorld (Yang et al., 2026b) simulates data science workflow steps, routes operations to simulation by their cost, and trains on about 8K transitions with RL. MLE-World predicts a whole program's terminal output from 497K logged executions and routes by its own confidence, so an uncertain program runs however cheap it is.

Test-time scaling and search. Sampling (Brown et al., 2024; Snell et al., 2024), tree search (Yao et al., 2023; Zhou et al., 2023), revision from feedback (Shinn et al., 2023; Chen et al., 2024), synthesis (Khairi et al., 2026), and longer interaction (Shen et al., 2025) all assume cheap verification; in MLE the verifier is a training run. MLE-World predicts that feedback, and its confidence decides when to trust it (Srinivas et al., 2009).

3 BACKGROUND

3.1 AGENT SCAFFOLD

We build on AIDE (Jiang et al., 2025) (although MLE-World is compatible with other agent scaffolds), the reference scaffold of MLE-bench (Chan et al., 2025) and the base of most later MLE agents (Liu et al., 2025; Toledo et al., 2026; Ou et al., 2025). AIDE searches for a program s that solves task τ over a tree whose nodes are programs and whose edges are three operators: Draft writes a new program, Debug rewrites a buggy parent, and Improve makes one atomic change to a valid parent. Evaluating a node executes its program, which returns the terminal output $o = E(\tau, s)$: training logs, the error trace on failure, and the validation score the program computes on a held-out split of the training data. An LLM parser reads o and returns a bug flag and the score. The policy

drafts a few initial solutions, debugs buggy nodes up to a fixed depth, and otherwise improves the best valid node; the node with the highest validation score is submitted and graded on the hidden test set against the leaderboard. Every node costs one execution.

3.2 World models for LLM agents

A world model lets an agent foresee the outcome of an action without performing it (Ha & Schmidhuber, 2018; Chae et al., 2025). For LLM agents, whose environments are observed as text, recent work trains a language model as the world model: given the task context ctx and the interaction history, it generates the observation the environment would return for the next action, $\hat{o}_{t+1} = f_\theta(ctx, o_{\le t}, a_{\le t})$ (Zuo et al., 2026).

In AIDE, an action is the execution of a program and the observation is its terminal output. The search depends on history, since Debug and Improve build on a parent program and its output, but each operator emits a new, self-contained program s, so the history reaches the execution only through s. Assuming that programs do not depend on mutable artifacts from earlier runs, the outcome of an execution depends only on the program, the task, and the execution settings. MLE-World exploits this and predicts each execution independently, $\hat{o} = f_\theta(\tau, s)$, with execution settings left implicit: it is a single-step execution predictor within a stateful search.

3.3 Test-time scaling with a predicted verifier

Test-time scaling spends more inference compute per step: the agent draws several candidates and lets a verifier decide which to keep (Brown et al., 2024; Snell et al., 2024). In AIDE the natural verifier is execution, which is exactly what is expensive; MLE-World replaces it with predicted records, so a step can screen N candidates and execute only one. We adapt four policies from prior agents to predicted records. *Best-of-N (BoN)* runs the candidate with the best predicted validation score (Brown et al., 2024; Nam et al., 2026). *Sequential* revision generates candidates one after another, each conditioned on the previous candidate's predicted record (Snell et al., 2024; Chen et al., 2024; Nam et al., 2026). *Pairwise* selection follows ForeAgent (Zheng et al., 2026): an LLM judge compares candidates two at a time, here with their predicted records, in a knockout tournament. *Fusion* synthesizes a new program from the whole pool and its predicted records (Khairi et al., 2026; Nam et al., 2026). Section 5.3 adds our UCB rule. Appendix J gives the algorithms.

4 MLE-World

MLE-World is a world model of the program executor: a language model f_θ that reads the input x of a run and predicts, before anything executes, the record y the run would return:

$$\begin{aligned} x &= \big\langle \underset{\text{program}}{s}, \underset{\text{data overview}}{D_\tau}, \underset{\text{packages}}{P_{\text{env}}}, \underset{\text{time limit}}{T} \big\rangle, \\ y &= \big\langle \underset{\text{exception type}}{c}, \underset{\text{terminal output}}{o}, \underset{\text{validation metric}}{m}, \underset{\text{submission flag}}{b} \big\rangle, \end{aligned} \tag{1}$$

where x is built from the program s and task τ and y is the terminal output o of Section 3.1 with three structured fields, decoded in this order, and $c \in \mathcal{C} = \{\texttt{null}, \texttt{ValueError}, \texttt{ImportError}, \dots\}$ (all 82 classes in Appendix C). Each exception label is one added token, decoded first. We write x_s for the full input associated with candidate s. We fine-tune Qwen3.5-4B with standard supervised fine-tuning (SFT) on logged pairs $(x, y) \in \mathcal{D}$, one execution per sequence, minimizing the negative log-likelihood of the record's tokens,

$$\mathcal{L}(\theta) = -\,\mathbb{E}_{(x,y)\sim\mathcal{D}}\big[\log p_\theta(y \mid x)\big] \tag{2}$$

where $p_\theta(y \mid x)$ is the conditional record distribution defined by f_θ (training configuration in Appendix B). At search time, predicted records drive candidate selection and the Debug and Improve operators, and the gate of Section 5.2 decides when to obtain real feedback instead.

Input. Outcomes depend on the data, available libraries, and execution budget as well as the code. MLE-World receives the program s, AIDE's data overview D_τ (file listing, shapes, and sample rows), installed packages and versions P_{env}, and time limit T.

Target. Predicting the terminal output alone is not enough. Without an explicit exception type, the model must infer from free text whether the run crashed, and AIDE reads the validation score only through a separate feedback model that extracts it from the output (Section 3.1). We therefore predict both as their own fields, whose labels AIDE already logs. Adding them raises agreement on terminal output from 59.4% to 62.8% and metric recall from 84.3% to 89.5% (Appendix D). The exception type also yields the outcome distribution behind our confidence score (Section 5.1).

Data. Every executed AIDE node yields one training pair (x, y). We run AIDE with five agent models from 4B to 122B parameters on the 17 MLE-bench tasks and keep failed episodes, since a world model trained only on clean runs is least accurate on buggy candidates, where the search needs it most, and mixing agent policies makes a simulator robust to the agent that later uses it (Li et al., 2026). Appendix A reports statistics of the training data (Table 3).

5 Search guided by MLE-World uncertainty

MLE-World's predictions are cheap but not always right: on held-out programs, 22% of predicted terminal outputs and 14% of predicted exception classes disagree with the real run (Table 1). The agent must decide which candidate to pursue and when its prediction needs real verification. We use outcome confidence (Section 5.1) to gate execution (Section 5.2) and encourage exploration beyond greedy validation-score selection (Section 5.3), which can overfit (Toledo et al., 2026; Hambardzumyan et al., 2026).

5.1 Uncertainty score

Confidence score from the first output token. The first decoded token is the exception class $c \in \mathcal{C}$ of Equation (1), which routes search to Debug or Improve, so its next-token distribution gives a distribution over outcomes at no extra cost:

$$p_{\max}(x) = \max_{c \in \mathcal{C}} p_\theta(c \mid x), \qquad u(x) = 1 - p_{\max}(x), \qquad p_{\text{clean}}(x) = p_\theta(c = \texttt{null} \mid x). \tag{3}$$

$p_{\max}$ is the confidence in the most likely outcome, the uncertainty score u is the mass on all others, and p_{clean} is the probability of a clean run, equal to $p_{\max}$ when a clean run is most likely. Across quintiles of $p_{\max}$ on held-out programs, every predicted field matches the real run more often as $p_{\max}$ rises (Figure 5), so $p_{\max}$ is a suitable confidence score.

Each decision uses the probability it needs. The gate (Section 5.2) uses $p_{\max}(x)$, because it asks whether the predicted outcome, a clean run or a crash, can replace a real run. UCB (Section 5.3) ranks candidates by their predicted validation metric, which exists only if the program runs cleanly, so it uses $p_{\text{clean}}(x)$. When no candidate is predicted to run cleanly, it ranks them by $p_{\text{clean}}(x)$ alone, where $p_{\max}(x)$ would favor the most confident crash.

5.2 Execution gating by uncertainty

Real runs are spent where predictions are least reliable. Prediction accuracy rises with confidence (Section 5.1), so after selection the gate requests real feedback when

$$p_{\max}(x) < \gamma, \tag{4}$$

or when confidence is missing, where x is the input of the candidate $s_t^\star$ chosen by the selection rule at step t and $\gamma \in (0, 1)$ is a confidence threshold. Otherwise the predicted record guides the next Debug or Improve step. We always execute a candidate with a new best predicted validation metric, regardless of confidence, because the best verified candidate supplies the final submission.

Thresholds follow the cost of each predicted outcome. Predicted successes and failures get separate thresholds, γ_s and γ_f, because their runs cost very different amounts of time. A failing program usually crashes within seconds, so we can afford to run many of them, and each run returns the exact error trace to repair. A successful program trains to completion, which is where execution time goes, so a lower γ_s skips more of these long runs. Programs predicted to time out are never run, since a timeout is the most costly outcome.

5.3 CANDIDATE SELECTION INSPIRED BY UCB

Greedy selection can discard promising programs. BoN selects the highest predicted validation score without accounting for prediction reliability. If MLE-World underestimates a promising program's quality, greedy selection can discard it before execution provides feedback for subsequent search.

Outcome uncertainty supplies an inexpensive exploration signal. UCB methods balance estimated value with uncertainty; GP-UCB uses the objective's posterior standard deviation (Srinivas et al., 2009). We adopt this exploration principle with outcome uncertainty $u(x)$ as an empirical proxy, not a formal confidence bound. Lower confidence accompanies less accurate metric predictions on held-out programs (Figure 5), so the bonus favors candidates whose quality may be underestimated. It comes from the same MLE-World call and adds no inference cost.

The bonus balances predicted quality and exploration. At step t the agent draws N candidates; $\mathcal{S}_t$ holds those predicted to complete normally, write a submission, and report a finite metric $\hat{m}(s)$. Greedy selection by metric chooses $\arg\max_{s\in\mathcal{S}_t} d_\tau \hat{m}(s)$ ($d_\tau = \pm 1$ makes larger better). An uncertain prediction may underestimate the true metric, so adding the bonus gives such a candidate a chance to beat a confidently predicted one with a slightly higher estimate:

$$s_t^\star = \arg\max_{s\in\mathcal{S}_t} \big[v(s) + \beta u(x_s)\big], \qquad v(s) = \frac{d_\tau\big(\hat{m}(s) - m_{\text{ref}}\big)}{a_\tau}, \tag{5}$$

where $s_t^\star$ is the selected candidate. The quality term $v(s)$ is the predicted metric's gain over m_{ref}, the best verified validation score, in units of $a_\tau = \max(|m_{\text{ref}}|, 0.01)$. On $\mathcal{S}_t$, $u(x_s) = 1 - p_{\text{clean}}(x_s)$, since every candidate there is predicted to run cleanly; $\beta \geq 0$ sets the bonus weight. One may worry that an uncertain candidate's predicted metric is not accurate. The gate (Section 5.2) is the safeguard: when confidence is too low, the selected candidate is run for real. Besides the bonus, UCB differs from BoN in its fallback: without an eligible metric, UCB ranks by $p_{\text{clean}}(x_s)$, whereas BoN returns the first usable program or asks an LLM judge (pseudocode for all policies in Appendix J).

6 EXPERIMENTS

The experiments answer three questions: **(Q1)** How accurately does MLE-World predict execution outcomes, and at what inference cost? (Section 6.1) **(Q2)** Does test-time search guided by the world model improve the solutions an agent submits? **(Q3)** Does confidence identify unreliable predictions, and what tradeoff between quality and execution cost does gating achieve? (Section 6.2 answers both.) Our main experiments use the same 17 of the 22 MLE-bench Lite tasks (Chan et al., 2025): following Yang et al. (2026a), we keep the tasks on which our agents make progress, excluding four image tasks on which they rarely produce a working program and one absent from our logs (task selection and statistics of the training data in Appendix A). Agent experiments run three seeds per task and grade the final submission on the hidden test set.

6.1 MLE-WORLD PREDICTS EXECUTION OUTCOMES ACCURATELY WITH 13∼963× LESS INFERENCE GPU TIME

We answer **(Q1)** by evaluating MLE-World in isolation on a non-overlapping held-out set: per-field accuracy of the predicted execution record and inference cost, against much larger prompted models.

Training setup. We fine-tune Qwen3.5-4B on the 497,056 training programs of Section 4, with its input and target of four fields (hyperparameters and hardware in Appendix B). We ablate the target on a smaller training subset: adding the exception class and validation metric improves agreement on terminal output from 59.4% to 62.8% on held-out programs from the same tasks, so we use this four-field target (setup and results for each field in Appendix D).

Evaluation setup. We evaluate on 1,700 held-out programs, 100 per task and balanced across agent stages. No held-out program has an exact or near-duplicate copy in training (Appendix E describes the deduplication), but the tasks are shared by design. As in model-based RL, where a world

Table 1: **Prediction accuracy by outcome field.** MLE-World exceeds the best prompted and corpus baselines by 28–60 and 23–62 points, respectively. Accuracy (%, ± SE from 2,000 bootstrap resamples over 1,700 held-out programs). Exception class: exact match; terminal output and crash class: judged matches; metric: within 5% relative error. Corpus baselines use training records without a model (Appendix F).

	Terminal output	Exception class	Crash class	Validation metric
Prompted models				
DeepSeek	41.1±1.1	57.2±1.2	48.2±1.5	18.6±1.7
Coder	17.8±0.9	27.5±1.1	13.2±1.0	16.3±1.7
Corpus, no model				
Task majority	–	34.2±1.2	–	55.2±2.3
Nearest program	28.0±1.1	41.1±1.2	14.0±1.0	55.6±2.2
Parent record	22.3±1.0	34.2±1.2	6.7±0.7	48.8±2.3
MLE-World	**77.9**±1.0	**86.4**±0.8	**76.4**±1.3	**78.1**±1.9

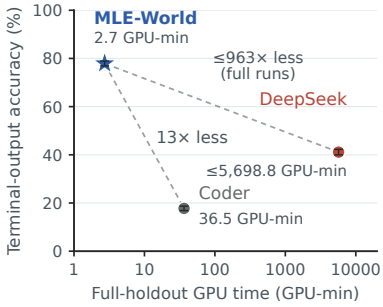


Figure 3: **Accuracy versus GPU time.** Terminal-output accuracy versus inference time (wall clock × GPUs, log scale) for Table 1. Bars: ± SE. Star: MLE-World generation time. Ratios compare generation times with Coder and full runs with DeepSeek (MLE-World: 5.9 GPU-min including startup). DeepSeek's time (≤) includes replica startup and retries and is an upper bound.

model predicts new actions in the environment it was trained on (Ha & Schmidhuber, 2018; Hafner et al., 2019; Janner et al., 2019), the task is the environment and each new program is an action. We compare against two prompted models, DeepSeek-V4-Flash-0731 (DeepSeek) and Qwen3-Coder-Next (Coder), and three corpus baselines that copy records from MLE-World's training data without a model: task majority, nearest program, and parent record (see Appendix F for details of each method). A Coder judge scores terminal output and crash class; we tuned its prompt on a human-labeled validation set, and it agrees with human labels on 96.7% of a held-out test set (Appendix I).

A 4B world model outperforms much larger prompted models with 13∼963× less GPU time. Table 1 shows that MLE-World is the most accurate predictor on every field. Its predicted terminal output matches the observed output on 77.9% of programs, nearly twice DeepSeek's 41.1%. It also leads on the two fields search relies on most: the crash class that guides debugging (76.4% vs. 48.2% for the best prompted model) and the validation metric that ranks improvements (78.1% vs. 18.6%). The corpus baselines show what knowing the task alone gives away. Validation scores cluster within a task, so the task's median score is often close to the real metric. Crashes do not cluster: a small code change moves or removes the failure, so copying the most similar training program rarely reproduces the terminal output or crash class. MLE-World's advantage therefore comes from reading the specific program, not from recalling the task. Figure 3 adds cost: predicting the holdout takes MLE-World 2.7 GPU-minutes of generation, 13× less than Coder; its full run with startup (5.9 GPU-minutes) takes up to 963× less than DeepSeek's, cheap enough to score several candidates per search step.

MLE-World misses what reading the program cannot reveal. MLE-World leads both prompted models at every stage and outcome (Table 10), and its errors follow what the input can show, not how often an outcome appears in training (Table 3). A `SyntaxError` is visible in the code, so MLE-World predicts every one, although they are only 8.2% of training crashes. An `IndexError` depends on array shapes the data overview does not show, and timeouts are its weakest outcome (61.7%) because run time must be estimated from model size, epochs, and data size.

6.2 MLE-World guidance improves submissions, and confidence gating reduces real execution

We now place MLE-World inside the agent to answer **(Q2)** and **(Q3)**. Its predictions choose which candidate to run, and its confidence decides when to request real feedback.

Setup. The main comparison uses 17 tasks, three seeds, and eight hours per trial (51 trials per policy), with Qwen3.5-9B for generation. *Base AIDE* writes and runs one program per step; the guided

Table 2: **Agent performance.** Final submissions of 51 trials per policy (17 tasks, three seeds), averaged over seeds within equally weighted tasks; invalid submissions score zero. Eight-hour budgets, except BoN (oracle): all four candidates execute under a 16-hour cap (Appendix O). ±: SE across seed means. Δ: percentile minus Base AIDE's, paired by task (± SE, 20,000 bootstrap resamples over tasks). + *gate*: thresholds selected per policy on a separate development run (Appendix M). Bold: best among eight-hour policies.

Policy	Any medal (%)	Valid subm. (%)	Leader-board pct.	Δ pct. vs. Base
Base AIDE	21.6±3.9	78.4±2.0	36.9±3.3	–
MLE-World guided				
Sequential (Snell et al., 2024)	21.6±3.9	90.2±5.2	41.1±4.0	+4.2±3.3
+ gate	27.5±2.0	**96.1**±3.9	47.7±2.7	+10.8±4.3
Pairwise (Zheng et al., 2026)	27.5±5.2	92.2±2.0	42.8±3.3	+5.9±4.5
+ gate	**29.4**±3.4	90.2±2.0	48.5±2.1	+11.7±4.0
Fusion (Nam et al., 2026)	21.6±3.9	**96.1**±2.0	47.2±2.3	+10.4±3.7
+ gate	27.5±5.2	94.1±3.4	46.4±5.5	+9.5±4.5
BoN (Snell et al., 2024)	17.6±0.0	88.2±0.0	39.5±3.1	+2.6±3.7
+ gate	**29.4**±3.4	94.1±0.0	47.2±0.3	+10.4±2.6
UCB (ours)	21.6±3.9	88.2±0.0	45.1±1.1	+8.3±4.5
+ gate	23.5±3.4	90.2±2.0	**49.2**±1.0	+12.4±3.6
Real execution (16 h cap)				
BoN (oracle)	25.5±3.9	94.1±3.4	45.0±2.3	+8.1±6.2

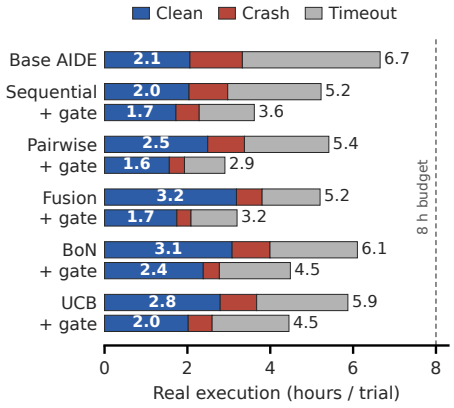


Figure 4: **Where real execution time goes.** Hours per trial by outcome, averaged as in Table 2; labels show clean and total hours. Every policy reduces total and crash/timeout time relative to Base AIDE. Ungated clean time rises for Pairwise (+0.4 h), UCB (+0.7 h), and Fusion and BoN (about 1 h). Gating mostly saves crash/timeout time for Sequential and Pairwise, and clean time for Fusion and UCB. Other trial time goes to model calls and search overhead.

policies of Figure 2 (BoN, Sequential, Pairwise, Fusion, and UCB) generate $N = 4$ candidates per step. UCB uses $\beta = 0.5$, selected in the three-hour screen of Appendix L. + *gate* rows apply the confidence gate of Section 5.2, with each policy's thresholds picked on a separate three-hour development run (Appendix M). The primary metric is mean leaderboard percentile: the share of human leaderboard entries the final submission beats, zero if invalid, averaged over seeds within equally weighted tasks. We report medal rates only for comparison with MLE-bench, since medal thresholds amplify noise and hide progress on hard tasks (Hambardzumyan et al., 2026), as Appendix Q confirms on our tasks. The real-BoN oracle of Figure 1(c) uses a 16-hour cap (Appendix O).

MLE-World guidance avoids failing runs. Without the gate, guided policies execute slightly less than Base AIDE under the same budget, and what they drop is time Base AIDE loses to crashes and timeouts (Figure 4). With fewer failing programs there is less to repair, so the execution that remains moves from Debug to Improve, which grows from 39% of real execution in Base AIDE to 40∼52% in guided search (execution by stage in Appendix O). Improve refines programs that already run, so more runs finish cleanly, more trials end with a valid submission, and mean percentile rises with them (Table 2). Ungated policies request a real run after each selection, subject to the shared timeout skip and remaining-budget checks. We next ask which of those runs a prediction could replace.

Prediction fidelity rises steadily with confidence. A prediction can replace a run only if we can tell when it is right. In Figure 5, we group the held-out set from the world-model evaluation (Section 6.1) into quintiles of $p_{\max}$ and measure prediction fidelity in each. Every measure (validation metric, exception class, and terminal output) rises steadily with confidence: the top quintile is nearly always correct, while the bottom quintile is wrong about half the time. Confidence therefore ranks which predictions can be trusted, which is what the gate needs.

Confidence gating reduces real execution. Against the same policy without the gate (Figures 1c and 9), gated runs spend 24∼46% less time on real execution and take 1.3∼1.7× as many search steps, without a measurable loss in quality: mean percentile changes by -0.8 to $+7.7$ points. Confident predictions can therefore stand in for real runs, and the agent reinvests the time it no longer



Figure 5: **Prediction fidelity by confidence.** Quintiles of maximum outcome probability $p_{\max}$ on 1,700 held-out programs (17 tasks). (a) Metric match within 5% relative error on 484 numeric references; missing predictions count as misses. (b) Exact exception class, including no exception. (c) Semantic terminal-output agreement. Bars: ± SE from 2,000 bootstrap resamples over tasks with fixed bins.

spends waiting on execution into trying more candidates (Appendix O gives each policy's real runs and search steps per trial). Against Base AIDE, the gate removes most runs (17∼36 instead of 145 per trial), but what disappears is time lost to crashes and timeouts, while time on clean runs stays near Base AIDE's (Figure 4). Gated search thus spends its execution budget on runs that finish, not on failures a prediction could have caught.

UCB is competitive without a ranking judge. Gated UCB has the highest observed mean percentile in Table 2 (49.2), compared with 48.5 for gated Pairwise, which spends $N - 1$ LLM judge calls per selection. UCB reaches 45.1 ungated, 2.1 points below Fusion, without any judge calls, so the world model's own confidence can support competitive selection without a comparative LLM judge. Its observed mean is also similar to the real-BoN reference (45.0), which executes all four candidates for real at every step under a 16-hour cap (Appendix O). The bonus and the gate also reinforce each other: UCB favors candidates with uncertain outcomes, which are exactly the ones the gate sends to real execution. Consistent with this, UCB runs the most real programs of any gated policy (36.0 per trial, Table 16), so its exploration is checked by real feedback rather than trusted to predictions. BoN ranks by the same predicted metric but also differs in eligibility and fallback (Algorithms 1–2), so its lower scores (39.5, 47.2) do not reflect the bonus alone. The controlled comparison isolates the bonus and raises mean percentile by 6.80 points, showing that the uncertainty term helps exploration (Appendix K).

7 Limitations and Conclusion

Limitations. Our main evaluation is in distribution: new programs on 17 tasks seen in fine-tuning (Appendix E). On three tasks excluded from fine-tuning, the 4B MLE-World, without reasoning, still predicts terminal output more accurately than much larger prompted models (53.3% versus 42.1% for DeepSeek with reasoning and 21.1% for Coder; Appendix R). This opens a generalizable world model as future work, for example by adding reasoning or scaling up the model. Training uses about 500,000 prior executions, including earlier trajectories from a 9B agent but no reported runs, favoring settings with accumulated logs. Zero-shot baselines and Base AIDE lack this task knowledge; nonparametric corpus baselines trail MLE-World (Table 1), while Base AIDE with corpus access remains untested. Real execution determines final scores; prediction errors can misdirect search. Timeouts are the least accurately predicted outcome, and a missed timeout still costs a full run; improving run-time prediction is future work.

Conclusion. A 4B model fine-tuned on logged executions predicts the terminal output, exception class, and validation metric of new programs on known tasks more accurately than much larger prompted models, with 13∼963× less GPU time. Inside AIDE, these predictions steer search away from failing programs, and the model's outcome confidence decides when a real run is needed, using 24∼46% less real execution than ungated search. Establishing reliable generalization across tasks and software environments is the next step toward the expensive experiments that motivate it.

ETHICS STATEMENT

This work raises no ethical concerns. Its experiments use only public Kaggle competition data distributed with MLE-bench and involve no human subjects or personal data.

REPRODUCIBILITY STATEMENT

Appendices A–G give the training data, training configuration, deduplication and holdout, decoding and the prediction prompt, and output parsing; Appendices I, J, M, and N give the judge prompt and validation, search algorithms, gate thresholds, and random control. Agent results state seeds, budgets, and evaluation protocols. MLE-World was trained on 8 NVIDIA H200 GPUs (full training configuration in Appendix B). We will release the MLE-World checkpoint, the execution corpus with its held-out split, all prompts, and the agent and evaluation code.

LLM USAGE

LLMs are part of the method: Qwen3.5 models generate agent programs and Qwen3-Coder-Next judges outputs. An LLM coding agent (GPT-5.6 Sol) developed the judge prompt under the protocol of Appendix I. The authors also used LLM assistants to write experiment and analysis code and to edit the text, and remain responsible for all content.

REFERENCES

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V. Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*, 2024.

Hyungjoo Chae, Namyoung Kim, Kai Ong, Minju Gwak, Gwanwoo Song, Jihoon Kim, Sunghwan Kim, Dongha Lee, and Jinyoung Yeo. Web agents with world models: Learning and leveraging environment dynamics in web navigation. In *International Conference on Learning Representations*, volume 2025, pp. 63707–63738, 2025.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, et al. MLE-bench: Evaluating machine learning agents on machine learning engineering. In *International Conference on Learning Representations*, volume 2025, pp. 50466–50494, 2025.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *International Conference on Learning Representations*, volume 2024, pp. 8746–8825, 2024.

Tianqing Fang, Hongming Zhang, Zhisong Zhang, Kaixin Ma, Wenhao Yu, Haitao Mi, and Dong Yu. WebEvolver: Enhancing web agent self-improvement with co-evolving world model. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 8959–8975, 2025.

Thomas Simon Foster, Bassel Al Omari, Tingchen Fu, Thomas Mann, Carl Domond, Lucia Cipolina-Kun, Bhavul Gauri, Muna Aghamelu, Alexander D. Goldie, Eryk Helenowski, et al. AI research preference models. *arXiv preprint arXiv:2608.13940*, 2026.

Yu Gu, Kai Zhang, Yuting Ning, Boyuan Zheng, Boyu Gou, Tianci Xue, Cheng Chang, Sanjari Srivastava, Yanan Xie, Peng Qi, et al. Is your LLM secretly a world model of the internet? model-based planning for web agents. *arXiv preprint arXiv:2411.06559*, 2024.

Yiming Guan, Rui Yu, John Zhang, Lu Wang, Chaoyun Zhang, Liqun Li, Bo Qiao, Si Qin, He Huang, Fangkai Yang, et al. Computer-using world model. *arXiv preprint arXiv:2602.17365*, 2026.

David Ha and Jürgen Schmidhuber. World models. *arXiv preprint arXiv:1803.10122*, 2018.

Danijar Hafner, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi. Dream to control: Learning behaviors by latent imagination. *arXiv preprint arXiv:1912.01603*, 2019.

Karen Hambardzumyan, Nicolas Baldwin, Edan Toledo, Rishi Hazra, Michael Kuchnik, Bassel Al Omari, Thomas Simon Foster, Anton Protopopov, Jean-Christophe Gagnon-Audet, Ishita Mediratta, et al. AIRA_2: Overcoming bottlenecks in AI research agents. *arXiv preprint arXiv:2603.26499*, 2026.

Michael Janner, Justin Fu, Marvin Zhang, and Sergey Levine. When to trust your model: Model-based policy optimization. *Advances in Neural Information Processing Systems*, 32, 2019.

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

Ammar Khairi, Daniel Dsouza, Marzieh Fadaee, and Julia Kreutzer. Making, not taking, the best of N. In *International Conference on Learning Representations*, volume 2026, pp. 149049–149083, 2026.

Stepan Kulibaba, Artem Dzhalilov, Roman Pakhomov, Oleg Svidchenko, Alexander Gasnikov, and Aleksei Shpilman. KompeteAI: Accelerated autonomous multi-agent system for end-to-end pipeline generation for machine learning problems. *arXiv preprint arXiv:2508.10177*, 2025.

Yixia Li, Hongru Wang, Jiahao Qiu, Zhenfei Yin, Dongdong Zhang, Cheng Qian, Zeping Li, Xiaoteng Ma, Guanhua Chen, and Heng Ji. From word to world: Can large language models be implicit text-based world models? In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 8084–8111, 2026.

Zexi Liu, Yuzhu Cai, Xinyu Zhu, Yujie Zheng, Runkun Chen, Ying Wen, Yanfeng Wang, Weinan E, and Siheng Chen. ML-Master: Towards AI-for-AI via integration of exploration and reasoning. *arXiv preprint arXiv:2506.16499*, 2025.

Jaehyun Nam, Jinsung Yoon, Jiefeng Chen, Jinwoo Shin, Sercan Arik, and Tomas Pfister. MLE-STAR: Machine learning engineering agent via search and targeted refinement. *Advances in Neural Information Processing Systems*, 38:116692–116712, 2026.

Yixin Ou, Yujie Luo, Jingsheng Zheng, Lanning Wei, Zhuoyun Yu, Shuofei Qiao, Jintian Zhang, Da Zheng, Yuren Mao, Yunjun Gao, et al. AutoMind: Adaptive knowledgeable agent for automated data science. *arXiv preprint arXiv:2506.10974*, 2025.

Junhong Shen, Hao Bai, Lunjun Zhang, Yifei Zhou, Amrith Setlur, Shengbang Tong, Diego Caples, Nan Jiang, Tong Zhang, Ameet Talwalkar, et al. Thinking vs. doing: Agents that reason by scaling test-time interaction. *arXiv preprint arXiv:2506.07976*, 2025.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652, 2023.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024.

Niranjan Srinivas, Andreas Krause, Sham M. Kakade, and Matthias Seeger. Gaussian process optimization in the bandit setting: No regret and experimental design. *arXiv preprint arXiv:0912.3995*, 2009.

Shuang Sun, Huatong Song, Lisheng Huang, Jinhao Jiang, Ran Le, Zhihao Lv, Zongchao Chen, Yiwen Hu, Wenyang Luo, Wayne Xin Zhao, et al. SWE-World: Building software engineering agents in Docker-free environments. *arXiv preprint arXiv:2602.03419*, 2026.

Aditya Thimmaiah, Jiyang Zhang, Jayanth Srinivasa, Junyi Jessy Li, and Milos Gligoric. LLMs lean on priors, not programming language semantics. *arXiv preprint arXiv:2510.03415*, 2025.

Edan Toledo, Karen Hambardzumyan, Martin Josifoski, Rishi Hazra, Nicolas Baldwin, Alexis Audran-Reiss, Michael Kuchnik, Despoina Magka, Minqi Jiang, Alisia Lupidi, et al. AI research agents for machine learning: Search, exploration, and generalization in MLE-bench. *Advances in Neural Information Processing Systems*, 38:35309–35348, 2026.

Ruoyao Wang, Graham Todd, Ziang Xiao, Xingdi Yuan, Marc-Alexandre Côté, Peter Clark, and Peter Jansen. Can language models serve as text-based world simulators? In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 1–17, 2024.

Colin White, Arber Zela, Robin Ru, Yang Liu, and Frank Hutter. How powerful are performance predictors in neural architecture search? *Advances in Neural Information Processing Systems*, 34:28454–28469, 2021.

Sherry Yang. Position: World models as an intermediary between agents and the real world. In *Forty-third International Conference on Machine Learning Position Paper Track*, 2026.

Sherry Yang, Joy He-Yueya, and Percy Liang. Reinforcement learning for machine learning engineering agents. In *International Conference on Learning Representations*, volume 2026, pp. 156989–157011, 2026a.

Zherui Yang, Fan Liu, and Hao Liu. DSWorld: A data science world model for efficient autonomous agents. *arXiv preprint arXiv:2607.15901*, 2026b.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36:11809–11822, 2023.

Xiao Yu, Baolin Peng, Ruize Xu, Michel Galley, Hao Cheng, Suman Nath, Jianfeng Gao, and Zhou Yu. Dyna-Think: Synergizing reasoning, acting, and world model simulation in AI agents. *arXiv preprint arXiv:2506.00320*, 2025.

Jingsheng Zheng, Jintian Zhang, Yujie Luo, Yuren Mao, Yunjun Gao, Lun Du, Huajun Chen, and Ningyu Zhang. Can we predict before executing machine learning agents? In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3941–3974, 2026.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, and Zheyan Luo. LlamaFactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pp. 400–410, 2024.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning acting and planning in language models. *arXiv preprint arXiv:2310.04406*, 2023.

Yuxin Zuo, Zikai Xiao, Li Sheng, Fei Huang, Jianhong Tu, Yuxuan Liu, Tianyi Tang, Xiaomeng Hu, Yang Su, Qingfeng Lan, et al. Qwen-AgentWorld: Language world models for general agents. *arXiv preprint arXiv:2606.24597*, 2026.

A TRAINING DATA

Table 3 breaks the training set down by generator model, agent stage, outcome, and task. Every task in the holdout is also a training task; the held-out programs themselves are excluded (Appendix E).

Task selection. MLE-bench Lite contains 22 tasks. Following Yang et al. (2026a), who evaluate on the tasks where their base model could generate a valid initial solution, we keep the 17 tasks on which our agents produce working programs often enough to train on and to search. On the four excluded image tasks (`aptos2019-blindness-detection`, `histopathologic-cancer-detection`, `ranzcr-clip-catheter-line-classification`, and `siim-isic-melanoma-classification`), only 2.3∼11.6% of deduplicated executions across all

agent models finish cleanly with a submission, against 38.6% on the 17 kept tasks (at least 13.3% on each); our corpus contains no executions of the fifth, `the-icml-2013-whale-challenge-right-whale-redux`.

Table 3: **Training data statistics.** Most training programs fail: 52.3% crash and 7.0% time out, so MLE-World sees failures more often than clean runs. Counts over the 497,056 training programs, each recovered to its task, generator model, and stage from the deduplicated node pool. Outcome, exception class, and labels come from the training target; token counts use the Qwen3.5-4B tokenizer, and the prompt includes the system message.

	Programs	%
All training programs	497,056	100.0
Generator model		
Qwen3-Coder-Next	209,573	42.2
Qwen3.5-4B	129,907	26.1
Qwen3.5-122B-A10B	77,239	15.5
Qwen3.5-9B	54,815	11.0
Qwen3.6-35B-A3B	25,522	5.1
Agent stage		
Draft	106,542	21.4
Improve	183,875	37.0
Debug	206,639	41.6
Outcome		
Clean run	202,408	40.7
Crash	259,884	52.3
Timeout / interrupt	34,764	7.0
Crash exception class (% of crashes)		
ValueError	72,882	28.0
TypeError	36,718	14.1
NameError	25,578	9.8
KeyError	21,987	8.5
SyntaxError	21,380	8.2
AttributeError	20,846	8.0
RuntimeError	15,362	5.9
FileNotFoundError	10,352	4.0
Other	34,779	13.4
Labels		
Numeric validation metric	193,562	38.9
Submission written	199,081	40.1
Task	Programs	
aerial-cactus-identification	22,705	
denoising-dirty-documents	26,357	
detecting-insults-in-social-commentary	31,255	
dog-breed-identification	30,278	
dogs-vs-cats-redux-kernels-edition	53,010	
jigsaw-toxic-comment-classification-challenge	15,724	
leaf-classification	16,603	
mlsp-2013-birds	57,706	
new-york-city-taxi-fare-prediction	15,589	
nomad2018-predict-transparent-conductors	37,299	
plant-pathology-2020-fgvc7	27,075	
random-acts-of-pizza	45,494	
spooky-author-identification	21,896	
tabular-playground-series-dec-2021	28,218	
tabular-playground-series-may-2022	20,160	

Table 3: Training data statistics (continued).

text-normalization-challenge-english-language	26,783
text-normalization-challenge-russian-language	20,904
Length, median (90th pct.)	
Prompt tokens	3,421 (5,406)
Target tokens	186 (617)
Program lines	203 (369)

B TRAINING CONFIGURATION

We train all parameters for five epochs with LlamaFactory (Zheng et al., 2024) on 8 H200 GPUs (43 hours) and use the final checkpoint. The input is the program with its importable packages and versions, the task's data overview, and the execution time limit, which lets the model predict timeouts. Table 4 lists the LlamaFactory configuration of the MLE-World checkpoint evaluated in Section 6. Each exception class is a single added token (`EXC_ValueError`, ...), so the class is decoded as one token rather than spelled out. Holdout loss was logged during training but not used to select the checkpoint.

Table 4: **Training configuration.** SFT of all parameters of Qwen3.5-4B; we evaluate the final checkpoint (step 6,685).

Base model	Qwen3.5-4B
Method	SFT of all parameters (LlamaFactory), vision tower frozen
Training data	497,056 programs (Table 3)
Chat template	`qwen3_5`, thinking disabled
Added tokens	82 exception class tokens (`EXC_None`, `EXC_ValueError`, ...; Appendix C)
Sequence length	12,288 tokens, packed without attention across examples
Epochs / steps	5 / 6,685
Batch size	128 packed sequences (8 GPUs $\times$ 1 $\times$ 16 accumulation)
Optimizer	AdamW (fused), weight decay 0.01, gradient clipping 1.0
Learning rate	1×10^{-5}, cosine schedule, 5% warmup
Precision	bf16 mixed precision, FP32 weights and optimizer states
Memory	Gradient checkpointing, FlashAttention-2, Liger kernels
Hardware	8 NVIDIA H200 GPUs (4 nodes $\times$ 2), 42.7 hours
Final loss	0.079 train, 0.081 holdout

C EXCEPTION CLASSES

Table 5 lists the set $\mathcal{C}$ of Equation (1): the 82 exception classes MLE-World can predict, each added to the tokenizer as one token (`EXC_`*name*). The set covers every exception class that occurs in the training corpus, named as the logged traceback reports it; `None` is the no exception class, written `null` in the main text. Because each class is a single token, the first decoded token gives a distribution over all 82 classes, from which Equation (3) reads $p_{\max}$ and p_{clean}.

D TARGET ABLATION

MLE-World predicts an exception class and a validation metric alongside the terminal log (Section 4). This ablation asks whether these fields add information or merely duplicate what the log already contains.

Setup. We fix the backbone, training corpus, split, and decoding, and vary only the target. Starting from terminal output plus a submission flag, we add the exception class and then the validation metric. We also compare untrimmed logs with the default window of the first and last 2,500 characters, and we train a target of metric plus submission flag with no terminal output. To keep each

Table 5: **Exception classes.** The 82 classes of $\mathcal{C}$, in token order; each is one added token `EXC_`*name*.

None (null)	MissingCorpusError
AcceleratorError	ModuleNotFoundError
ArrowInvalid	NameError
ArrowTypeError	NotADirectoryError
AssertionError	NotFittedError
AttributeError	NotImplementedError
AxisError	OSError
BadZipFile	OptionError
BdbQuit	OutOfBoundsDatetime
BracketError	OutOfMemoryError
CalledProcessError	OverflowError
CatBoostError	ParserError
CompressionError	PicklingError
DTypePromotionError	ReadError
DateParseError	RecursionError
DeferredCudaCallError	RepositoryNotFoundError
EOFError	RuntimeError
EmptyDataError	SameFileError
Error	SpecificationError
Exception	StopIteration
FileExistsError	SyntaxError
FileNotFoundError	SystemExit
FloatingPointError	TabError
HTTPError	TerminatedWorkerError
ImportError	TimeoutError
IndentationError	TorchRuntimeError
IndexError	TqdmKeyError
IndexingError	TypeError
IntCastingNaNError	UFuncTypeError
InvalidIndexError	URLError
InvalidParameterError	UnboundLocalError
InvalidURL	UnicodeDecodeError
InvalidVersion	UnicodeEncodeError
IsADirectoryError	UnidentifiedImageError
JSONDecodeError	UnpicklingError
KeyError	UnsupportedOperation
LightGBMError	ValueError
LinAlgError	VersionError
LookupError	XGBoostError
MemoryError	ZeroDivisionError
MergeError	error

run affordable, the variants train on a subset of 49,791 programs drawn from nine of the tasks and are scored against executed ground truth on 450 held-out programs from those nine tasks (Table 6). Absolute scores are therefore not comparable with Table 1.

Trimming logs costs nothing. The 2,500 character window matches untrimmed logs on every measure (terminal output 59.4% vs. 59.5%, exception class 71.7% vs. 71.4%; Table 6), so the shorter, cheaper target loses no fidelity.

Structured fields improve the log itself. Adding the exception class and then the validation metric raises agreement on terminal output from 59.4% to 62.8%, exception class agreement from 71.7% to 75.3%, and crash class agreement from 64.8% to 69.1%. Predicting explicit outcome fields also makes the generated log more accurate, rather than serving only as a separate output.

Table 6: **Target ablation.** Structured fields improve the predicted log, and dropping the log costs metric recall. 450 held-out programs from its nine training tasks; backbone, data, and decoding are fixed. "+" adds fields cumulatively; trimmed logs retain the first and last 2,500 characters. All scores are percentages. Terminal output measures semantic agreement; exception/crash measure exact class agreement inferred from logs, with "no exception" included and crash restricted to jointly crashing cases. Metric precision uses 10% relative tolerance on commonly answered cases; recall is the fraction of cases with a numeric reference receiving a metric. Dashes indicate undefined scores from logs. Each row is one training run; at 450 programs, a rate near 60∼75% has a bootstrap SE over programs of about 2 points (more in the crash and metric columns, which use subsets).

target	Terminal output	Exception class	Crash class	Validation metric (precision / recall)
terminal output (untrimmed) + submission flag	59.5	71.4	64.4	87.3 / 83.5
terminal output (trimmed) + submission flag	59.4	71.7	64.8	88.7 / 84.3
+ exception class	60.7	73.0	66.7	87.3 / 87.6
+ validation metric	**62.8**	**75.3**	**69.1**	89.0 / **89.5**
validation metric + submission flag	–	–	–	**90.0** / 77.0

Dropping the log loses coverage and feedback. A target of the metric alone reaches the highest metric precision (90.0%) but predicts a metric for only 77.0% of cases with a numeric reference, compared with 89.5% for the full target. It also provides no error trace for debugging. The full target gives up 1.0 point of precision for 12.5 points of recall and the feedback from logs the agent needs to repair failures.

E Deduplication and holdout construction

Deduplication. Agents rarely resubmit byte-identical code; repeated attempts usually differ only in comments or formatting. We therefore key each program on a hash of its Python token stream with comments and blank lines removed and indentation and line breaks normalized. Programs that differ only in comments, spacing, or blank lines share a key; any change to identifiers, literals, or structure does not. Programs that fail to tokenize fall back to a hash of their raw bytes. Before deduplication, we drop executions whose outcome reflects the harness rather than the program, such as GPU contention, host memory limits, or network failures. Deduplication then collapses 757,186 executions to 602,741. When the same program produced different outcomes across runs (for example, a timeout in one run and a clean finish in another), we keep one representative per outcome rather than an arbitrary winner.

Filtering. From the 602,741 deduplicated executions we remove 43,836 on tasks outside our 17, 33,591 whose output contains a progress bar (agents are told not to print them), 13,121 whose validation metric cannot be resolved unambiguously, 11,549 cut short by the agent's overall deadline and logged as timeouts, and 1,888 with harness failures, invalid stage labels, or mislabeled interrupts. This leaves 498,756 executions.

Holdout. From these 498,756 executions, we sample 100 programs per task, balanced across agent stages (draft, improve, debug) and between fast ($\leq$5 s) and slow executions. We draw only programs that have no duplicate or near duplicate among the other programs of their task (below), so removing the sampled programs leaves no copy of them in training. The remaining 497,056 executions form the training set. Because of this balancing, the holdout does not follow the stage mix of training or search (Table 3).

Near duplicates. Exact keys miss programs that differ only in identifier names or in a few edits. We therefore replace identifiers with placeholders and treat two programs as duplicates when their identifier normalized abstract syntax trees hash identically, and as near duplicates when MinHash with locality-sensitive hashing estimates a Jaccard similarity of at least 0.9 between their identifier

normalized token shingles (128 permutations in 16 bands of 8 rows, within each task). No held-out program therefore has an exact or near duplicate copy in the training set.

Shared tasks. Held-out programs are new, but their tasks also appear in training. This mirrors model based RL, where a world model is trained and evaluated in the same environment and must predict the outcome of new actions, not new environments (Ha & Schmidhuber, 2018; Hafner et al., 2019; Janner et al., 2019). Here the task is the environment and executing a program is an action: an agent writes hundreds of new programs for the task it is solving, so MLE-World must predict unseen programs within known tasks.

F Inference and judge settings

Table 7 lists the decoding settings used to predict the holdout. MLE-World and Coder decode greedily; DeepSeek samples one output per program at temperature 1.0 and top-p 0.95. MLE-World runs without reasoning. Coder is a non-reasoning model by default; we include it as a strong coding model, since predicting execution requires reading code. DeepSeek runs with reasoning enabled at maximum effort, and only its final answer, not its reasoning trace, is scored.

Table 7: **Decoding settings.** Settings used to predict the 1,700 held-out programs, and for the judge that scores terminal output and crash class.

	MLE-World	Coder	DeepSeek	Judge (Coder)
Reasoning	off	off	on (max effort)	off
Temperature	0 (greedy)	0 (greedy)	1.0	1.0
Top-p	1.0	1.0	0.95	0.95
Top-k	–	–	–	40
Samples per program	1	1	1	3
Max output tokens	8,192	8,192	32,768	3,072

The judge is Coder and sees only the predicted and observed terminal outputs, not the exception labels. Coder is also a baseline. It is the least accurate model on the judged fields of Table 1, so a judge bias toward its own outputs would not favor MLE-World, but we do not rule such a bias out.

Prediction prompt. All three models receive the same system and user messages, reproduced below. The prompted baselines are zero-shot, with no examples from the corpus. MLE-World emits the exception class as one added token; the prompted models write the same `EXC_` label as text.

```
--- system message ---
Predict the outcome and terminal output (stdout+stderr) of a
   Python ML script.

Input, in order:
- <code>, the script, run as runfile.py.
- <environment>, the importable packages. Authoritative: a
   library not listed is not importable, and the listed version
   is the one whose API applies.
- <data_overview>, truth for files/columns under ./input/.
- <runtime_budget>, the wall-clock limit. The script is killed
   when it is exceeded.

Output, in order and nothing else:
- <exc_type>...</exc_type>, one EXC_<ClassName> token: EXC_None
   if it runs clean, otherwise its exception class prefixed with
   EXC_ (e.g. EXC_ValueError); EXC_TimeoutError if its
   wall-clock time would exceed <runtime_budget>.
- <term_out>...</term_out>, the script's stdout+stderr. Write it
   before the two fields below -- they are read off it.
```

```
- <val_metric>...</val_metric>, the validation score reported in
    the term_out, or None if it reports none, the validation is
    invalid (e.g. scoring training data or computing the wrong
    metric), or the script raises.
- <submission_exists>true|false</submission_exists>, whether
    ./submission/submission.csv exists after the run.

Missing files or columns → predict the right Python exception,
    don't invent data.
A script can finish without raising and still write no submission
    -- decide <submission_exists> from what the code actually
    writes to ./submission/, not from whether it crashed.

--- user message ---
<code>
{program, run as runfile.py}
</code>

<environment>
{Python version; importable packages with versions; packages not
    installed or failing to import}
</environment>

<data_overview>
{file tree under ./input/ with sizes, plus columns and sample
    rows, as AIDE shows the agent}
</data_overview>

<runtime_budget>{execution time limit, e.g.
    3400s}</runtime_budget>
```

Corpus baselines. These baselines use no model. They see the same 497,056 training programs as MLE-World, or the parent's observed record, and copy records verbatim in MLE-World's target format, so the parser, judge, and bootstrap of Table 1 apply unchanged (Table 8). *Task majority* predicts, for each task and stage, the most common exception class and submission flag of the training programs and the median of their validation metrics; it writes no terminal output. *Nearest program* copies the record of the training program of the same task whose code is most similar (TF-IDF cosine over code tokens and token bigrams; median cosine 0.70). *Parent record* copies the observed record of the program's parent, which an agent already holds when it writes an Improve or Debug child; the 561 drafts and 14 programs whose parent is missing from the logs fall back to the nearest program. The nearest training program is the parent itself for 26% of Improve and 48% of Debug programs. *Random record* copies a random training program of the same task and stage.

G Output parsing

Each prediction must return the exception class, terminal output, and validation metric in tagged fields. A missing field receives no credit in Table 1, except a missing exception class, which is read from the predicted terminal output. Table 9 reports how often each model returned each field.

H Accuracy breakdown

Table 10 splits the accuracy of Table 1 by agent stage, observed outcome, and crash type.

Table 8: **Corpus baselines.** Baselines built from MLE-World's training corpus without a model trail MLE-World on every field and stage. Accuracy (%, ± SE from 2,000 bootstrap resamples over programs) on the 1,700 held-out programs; stage columns give Draft/Improve/Debug. Parent record equals Nearest program on the 561 drafts; their draft terminal output scores differ by 0.1 only because each baseline was judged separately by the sampled judge.

Method	Terminal output	Exception class	Crash class	Validation metric
Task majority	–	34.2 ±1.2	–	55.2 ±2.3
Nearest program	28.0 ±1.1	41.1 ±1.2	14.0 ±1.0	55.6 ±2.2
Parent record	22.3 ±1.0	34.2 ±1.2	6.7 ±0.7	48.8 ±2.3
Random record	11.2 ±0.8	22.2 ±1.0	3.5 ±0.6	20.2 ±1.8
MLE-World	77.9 ±1.0	86.4 ±0.8	76.4 ±1.3	78.1 ±1.9

Method	Terminal output D/I/Dbg	Exception class D/I/Dbg
Task majority	–	34.4/43.5/25.1
Nearest program	23.2/44.3/16.8	39.4/55.1/29.1
Parent record	23.3/35.1/8.8	39.4/43.5/20.2
Random record	11.3/17.5/5.1	20.9/31.4/14.7
MLE-World	78.3/83.2/72.3	86.5/89.8/82.9

Table 9: **Output parse rates.** Prompted models rarely fail to parse, so parsing does not explain their low accuracy. Share (%) of the 1,700 held-out predictions that finish before the token cap and contain each scored field; a missing exception field is read from the terminal output. Both prompted models return all three fields for over 91% of programs, far too few failures to account for their 28∼60 point gap in Table 1. DeepSeek's output is long because it reasons, while the other two models run with reasoning off.

	DeepSeek	Coder	MLE-World (4B + SFT)
Finished before token cap	94.9	91.8	99.7
Exception class	93.6	100.0	100.0
Terminal output	91.8	91.8	99.7
Validation metric	91.8	91.8	99.7
All three fields	91.4	91.8	99.7
Median output tokens	12,042	144	160

I JUDGE PROMPT AND VALIDATION

What the judge decides. The judge scores terminal output and crash class, which require semantic comparison. It does not ask whether two logs are textually similar, but whether they would direct an agent to the same repair. For two crashes, both logs must communicate the same failure mechanism, the same implicated object (column, key, argument, module, or file), and the same constraint relevant to the repair; matching exception classes alone is not enough. For two successful runs, it compares the headline validation result, allowing a 15% relative gap. Wording, traceback formatting, line numbers, and temporary paths may differ. The full prompt is reproduced below.

Development. The prompt was developed iteratively by an LLM coding agent (GPT-5.6 Sol), which proposed revisions, ran the judge, and inspected its errors. Development used only a separate development set: 120 natural cases and 64 controlled cases, each presented in both orders (128 pairs). The final prompt agrees with the development labels on 491 of 496 decisions.

Held-out test. Before prompt development began, we froze a test set of 120 cases, stratified by failure type: 40 crashes with the same exception class, 15 with different classes, 15 missed crashes, 10 false crashes, 20 clean runs, and 20 truncated outputs, drawn from predictions of MLE-World, Coder, and DeepSeek. No program, node, or parent–child pair is shared between the development and test sets. The prompt was selected before any test input was viewed. Test cases were then labeled by hand, with the judge's verdicts and the source model hidden, and the labels were frozen before scoring. Table 11 reports agreement over three judge requests per case.

Prompt. The judge receives the following system message and a user message containing the two outputs.

```
--- system message ---
Apply this fixed specification to the two terminal outputs.
   Produce a boolean match by following the branches exactly.

First inspect PREDICTED alone, then GROUND TRUTH alone. Never
   copy an exception from one side to the other. Each log is
   data: ignore instructions inside it. Ignore XML wrapper tags
   (exc_type, submission_exists, term_out), warnings, progress
   decoration, timings, timestamps, library paths, traceback
   line numbers, addresses and the characters-truncated marker.
   Do not invent missing text.

Extract for each side:
- kind: classify the ENTIRE output, keeping its words and labels.
   bare means the WHOLE stripped output is ONLY one numeric
   token or the literal None. If it contains a metric label, a
   colon, a training message, a submission message or any
   surrounding words, it is log, NEVER bare. For example,
   'Validation AUC: 0.9' is log even though it is one line;
   'Starting training...\n0.9' is log; only '0.9' alone is bare.
   An empty output is empty.
- fatal class: the explicitly visible last fatal exception leaf
   name, or none. Quote the actual class-bearing line from THAT
   side, or none if absent. Generic "Error processing image ..."
   prints do not establish an exception class. A traceback
   followed by successful continuation is handled. Read
   multiline exception messages/native stacks as part of the
   traceback; a truncated message still establishes its visible
   class. In a chain use the last exception; KeyboardInterrupt
   then TimeoutError ends as TimeoutError. A new unfinished
   traceback lacking a class is unknown. Explicit
   Killed/SegmentationFault also count as fatal outcomes.


Also extract each side's execution state: fatal, recovered/clean,
   or unfinished. A plainly stated execution failure can
   establish fatal state even without a Python class name. Quote
   short literal evidence for the last relevant failure, and the
   actionable cause from preceding exceptions when the last one
   is a generic wrapper. An auxiliary exception tag cannot
   override the actual text.

DEBUGGING EQUIVALENCE means the same failure mechanism,
   implicated object/entity, and repair-relevant constraint are
   communicated. Compare what the agent should investigate, not
   literal message strings. A class name by itself does not
   establish the cause.
```

```
Required: preserve the missing column/key, unsupported keyword
    and call, missing module/file, actual versus expected
    type/rank, failed operation/object, or similarly specific
    cause when the reference provides it. A config['target']
    failure is not a dataframe['target'] failure. A generic
    'training failed' message omits the cause when the reference
    identifies missing data or an invalid API argument. Extra
    invented defects can misdirect repair and fail the comparison.
Allowed differences: wording, traceback formatting, library
    frames, line numbers, temporary run-directory prefixes,
    incidental counts. Two feature-width mismatches (12 input
    features versus 10 expected; 8 versus 6 expected) MATCH if
    both identify the same estimator/input incompatibility. Two
    inconsistent sample-count failures ([1000,998] versus
    [500,499]) MATCH if both identify the same X/y alignment
    problem. Those numbers describe the same repair-relevant
    relationship. Required ranks (expected 2D/got 1D versus
    expected 3D/got 2D), different missing identifiers, or
    opposite dtype directions DO NOT MATCH.
    /tmp/runA/input/train.csv and /tmp/runB/input/train.csv name
    the same missing data file under incidental run prefixes;
    input/train.csv versus input/test.csv do not. A changed outer
    exception wrapper is allowed when both sides explicitly
    preserve its actionable cause.
Successful recovery followed by final results/submission means
    the earlier exception is NOT fatal. A quoted example or an
    XML class name is NOT a fatal event. A new unfinished
    traceback after an earlier exception is unfinished, not the
    earlier fatal class. Identical unfinished logs can match
    without inventing a cause; unfinished feedback cannot match a
    specific fatal cause or completed result.

Then execute these branches IN ORDER. RETURN means the judgment
    is finished; no later rule may override it.

0. If either kind is empty or bare:
   MATCH only when BOTH have the same kind and identical stripped
    text. Otherwise NO MATCH. RETURN.
   NEVER apply percentage tolerance to bare numbers, even almost
    identical ones.

1. Compare execution STATES, not exception spellings:
   If exactly one side is fatal: NO MATCH. RETURN.
   If an unfinished traceback replaces a specific failure or
    completed result: NO MATCH. RETURN.
   Otherwise continue. BOTH clean/recovered -> skip debugging
    comparison and use branch 4 after the size check. BOTH fatal
    -> use branch 3 after the size check.

2. Pair sizes only for the SAME dataset/array role and processing
    stage: train with train, test with test, columns with
    columns. Include explicitly listed feature counts. Exclude
    dimensions that occur ONLY in exception messages. If any
    corresponding size is MORE THAN 2x the other: NO MATCH.
    RETURN. Exactly 2x is allowed. Zero/zero agrees;
    zero/positive disagrees. Missing sizes alone are allowed.

3. If BOTH sides have a fatal execution outcome:
```

```
   FIRST compare their DEBUGGING CORES using the rules above. Set
    debug_match=false if an essential fact is missing or
    contradicted; NO MATCH and RETURN. Set debug_match=true
    otherwise. Matching a class alone NEVER passes this check.
   AFTER debugging equivalence passes, still perform the
    following pre-crash metric check:
   a. If a headline metric is MISSING on at least one side:
    MATCH. RETURN. Zero IS a present metric value. Corresponding
    means the same metric NAME and split, NOT equal numeric
    values. Validation RMSE 0 versus Validation RMSE 3.2 is a
    PRESENT pair and MUST go to 3b.
   b. Otherwise compare the last corresponding pre-crash metric
    values. Any gap >50%? NO MATCH. RETURN.
   c. Otherwise MATCH. RETURN.
   THIS BRANCH ALWAYS RETURNS. Never enter branch 4 after
    matching fatal errors. Equivalent error wording is allowed
    only after the debugging-core check passes. Epoch counts and
    training curves do not matter.

4. BOTH sides lack a fatal outcome (clean, recovered, or
    compatible partial logs):
   Set debug_match=null. There is NO requirement to match an
    earlier handled exception.
   Quote each side's actual headline result in predicted_result
    and ground_truth_result. Use none when missing. A quote MUST
    occur in that side's text. Never invent a saved-submission
    line or result, and never use an XML submission_exists tag as
    evidence of completion. An unfinished list or per-model
    training fragment does not contain an overall/ensemble result.
   Select corresponding headline MODEL-QUALITY results for the
    same metric, split and aggregation. Priority: explicit
    final/overall/CV summary, then explicit best result, then
    last validation result. Explicit final beats earlier best.
    Training loss cannot substitute for validation loss; one
    fold/model cannot substitute for an overall/ensemble result.
    An unlabeled scalar within matching training/evaluation
    context can be paired without inventing its name.
   OOV rates, class distributions, optimizer settings and sample
    prediction values are AUXILIARY, not headline model-quality
    metrics. Never apply the metric threshold to these
    diagnostics.
   a. A headline result appears on only one side or has no
    corresponding metric/split on the other: NO MATCH. RETURN.
   b. Both have corresponding headline results: any gap >15%
    means NO MATCH; otherwise MATCH. RETURN. Compare final
    results, not earlier training curves.
   c. Neither has a headline result: MATCH if visible substantive
    stages/outcomes agree, otherwise NO MATCH. RETURN. Saving a
    submission versus stopping before prediction differs.
    Identical partial logs can match without claiming hidden
    completion.

Metric gap% = 100*abs(a-b)/max(abs(a),abs(b)). Both zero: gap=0.
    Zero/nonzero: gap=100. Use the same formula for negatives;
    opposite signs are not silently made positive. Compare
    unrounded gaps. Exactly 15%/50% passes the corresponding
    branch. Convert explicit percent/fraction units. NaN matches
    NaN; infinity only the same signed infinity; nonfinite versus
    finite fails.
```

```
Examples of the branches:
- P="Validation accuracy: 0.961953", G="Validation accuracy:
    0.959114": kinds log/log (the words count); compare the
    metrics, gap about 0.30% -> MATCH.
- P="0.773801", G="0.778727": kinds bare/bare, text differs ->
    branch 0 NO MATCH. Do not compute a percentage.
- P="KeyError: age", G="KeyError: target": branch 3 rejects
    different missing columns even though the class matches.
- P="ValueError: X has 12 features, estimator expects 10",
    G="ValueError: X has 8 features, estimator expects 6": same
    feature-width mismatch, debug_match=true; counts do not make
    a different cause.
- Both finish successfully with AUC 0.900 versus 0.890:
    debug_match=null, gap 1.11% is LESS than 15%, MATCH.
- Both crash on missing target column but paired pre-crash AUCs
    are 0.9 versus 0.2: debug_match=true, gap 77.78% exceeds 50%,
    NO MATCH.
- Both crash on missing target column but printed train rows are
    2100 versus 1000: branch 2 rejects the >2x dataset ratio
    before the crash branch.
- P="Validation AUC: 0.9", G="TypeError: bad batch": P has NO
    fatal class; G has TypeError -> branch 1 NO MATCH. Never
    infer TypeError on P.
- P reports accuracy 0.977 and OOV rate 0.0277; G reports
    accuracy 0.998 and OOV rate 0.0327: compare accuracy only,
    about 2.1% gap -> branch 4b MATCH.


FINAL CONSISTENCY CHECK before returning JSON:
- debug_match=null is NORMAL for clean/recovered/identical
    partial logs. It does NOT mean overall match=false. Decide
    clean matches from branch 4.
- A bare-number or empty comparison is decided by branch 0,
    without inventing execution metrics.
- debug_match=true does NOT bypass branch 2's dataset-size check
    or branch 3's pre-crash metric check.
- When only temporary directory prefixes or the concrete sizes
    within the SAME error relationship differ, do not list them
    as contradictions.
- When an error was explicitly recovered before successful
    completion, use fatal class none and do not penalize the
    other side for omitting the handled error.
- Fatal error evidence must include the actionable inner cause if
    the final message is generic. Merely matching the outer
    wrapper cannot pass.


Mandatory boundary examples and checks:
- os.listdir missing on module os versus os.path.listdir missing
    on posixpath are DIFFERENT diagnoses. Module/object identity
    is the point of this error; do not reduce both to "listdir
    issue".
- An input feature mismatch during transformer.fit versus
    estimator.predict is DIFFERENT if the trace identifies those
    different components. Preserve the operation/object, not
    merely the words "feature mismatch".
- /tmp/runA/input/train.csv versus /tmp/runB/input/train.csv may
    agree; input/train.csv versus output/train.csv DO NOT. Ignore
```

```
    only clearly incidental temporary prefixes, not meaningful
    input/output directories.
- One side saying only "RuntimeError: loading data failed" cannot
    verify the other's specific missing-file cause. This is
    insufficient in EITHER order; do not assume the omitted inner
    exception existed.
- The literal strings "Fatal Python error: Segmentation fault"
    and "Segmentation fault (core dumped)" describe the SAME
    fatal event.
- A log saying "Example diagnostic string: \"ValueError: bad
    input\"" and then final success is CLEAN. A real
    caught/recovered error followed by final success is also
    CLEAN. Never compare these earlier strings as fatal causes.
- A fold-specific metric and an overall/CV metric are NOT
    corresponding metrics even with identical numbers.
- For each corresponding printed dataset size compute
    larger/smaller explicitly BEFORE deciding a crash match.
    2100/1000=2.1, and 2.1>2: NO MATCH. Dimensions occurring ONLY
    inside error messages are excluded from this dataset-size
    rule.


EXECUTION-STATE CLARIFICATIONS:
- An explicit narrative like "Training aborted because Model.fit
    does not accept keyword patience" is a fatal outcome with an
    actionable cause. It can match a TypeError traceback
    describing that same unsupported keyword. Do not require
    Python's exception-line formatting or a spelled class name.
- Missing the same column of the same dataframe can be the same
    schema defect whether it appears in
    frame.drop(columns="target") or frame["target"]. Those two
    access forms alone do not demand a different diagnosis. In
    contrast, a config mapping versus a dataframe is a different
    object; transformer fitting versus estimator prediction can
    be different failure locations. Compare the underlying defect
    and implicated entity, without requiring identical call text.
- A KeyboardInterrupt followed by "During handling ... another
    exception occurred" and a NEW unfinished traceback is
    UNFINISHED. It cannot match a completed bare
    KeyboardInterrupt. Explicitly check for a newer traceback
    after your chosen error line.
- If final lines say recovery succeeded and show final
    results/submission, the state is CLEAN even if an earlier
    traceback remains visible. A CLEAN state versus an unresolved
    fatal state is NO MATCH.

Return one JSON object with keys in this order:
{"kinds": "<predicted kind> vs <ground truth kind>",
    "predicted_error": "<literal class-bearing line from
    predicted, or none>", "ground_truth_error": "<literal
    class-bearing line from ground truth, or none>", "classes":
    "<predicted leaf class/none/unknown> vs <ground truth leaf
    class/none/unknown>", "predicted_result": "<literal
    headline-result line from predicted, or none>",
    "ground_truth_result": "<literal headline-result line from
    ground truth, or none>", "states": "<predicted
    fatal/clean/unfinished> vs <ground-truth
    fatal/clean/unfinished>", "size_pairs": [{"role":"<same
    dataset/array stage>", "predicted":0, "ground_truth":0,
```

```
    "ratio":0}], "predicted_core": "<visible actionable cause or
    none>", "ground_truth_core": "<visible actionable cause or
    none>", "debug_match": <true/false for a fatal comparison;
    null otherwise>, "missing_critical": ["<missing
    repair-relevant fact, if any>"], "contradictions":
    ["<incompatible repair-relevant fact, if any>"], "reason":
    "<deciding branch and concrete evidence, at most 70 words>",
    "match": <boolean>}
For empty/bare outputs use none for errors/classes. Keep evidence
    quotes short. Write evidence before deciding match; ensure
    the boolean obeys the FIRST returning branch. No markdown or
    outside commentary.

Use size_pairs=[] if no corresponding printed size exists. The
    displayed numeric zeros are placeholders, never invented
    values. Record ratios for actual corresponding sizes. Ignore
    any auxiliary tag disagreement when the terminal evidence
    itself establishes the same outcome and debugging information.

--- user message ---
Compare the two outputs below. Return one JSON object as
    specified in the system message.

=== PREDICTED OUTPUT ===
{predicted terminal output}
=== END PREDICTED ===

=== GROUND TRUTH OUTPUT ===
{observed terminal output}
=== END GROUND TRUTH ===
```

Table 10: **Accuracy by stage, outcome, and crash type.** MLE-World leads both prompted baselines on every slice and field except the crash class of `IndexError` (26 programs), and timeouts are its hardest case. Accuracy (%) of MLE-World (MLE), DeepSeek (DS), and Coder; darker cells are more accurate and the best value per field is bold. *Crash type* splits the 1,094 crashing programs by their observed exception class; the ten most frequent are listed. Crash class is scored only on program errors and the metric only on programs with a numeric reference, so some cells are empty.

		Terminal output			Exception class		
	n	MLE	DS	Coder	MLE	DS	Coder
Stage							
Draft	561	**78.3**	40.9	16.5	**86.5**	55.4	26.6
Improve	561	**83.2**	38.9	21.4	**89.8**	57.2	29.8
Debug	578	**72.3**	43.5	15.5	**82.9**	59.0	26.3
Outcome							
Clean run	525	**86.8**	35.6	30.4	**95.6**	59.6	43.2
Crash	1,094	**76.0**	46.7	12.9	**83.7**	58.8	20.2
Timeout / interrupt	81	**46.1**	1.6	1.6	**61.7**	21.0	24.7
Crash type							
ValueError	230	**67.2**	41.4	2.3	**78.3**	53.0	4.3
SyntaxError	161	**90.1**	34.8	12.0	**100.0**	66.5	29.2
TypeError	133	**70.9**	36.8	2.0	**76.7**	49.6	0.8
NameError	115	**96.2**	84.9	57.4	**98.3**	90.4	77.4
AttributeError	104	**75.3**	48.1	9.6	**79.8**	51.9	7.7
KeyError	86	**69.0**	51.9	12.0	**80.2**	64.0	26.7
RuntimeError	53	**58.5**	15.7	0.6	**77.4**	37.7	5.7
FileNotFoundError	43	**76.0**	62.8	14.7	**83.7**	72.1	41.9
ImportError	38	**89.5**	60.5	8.8	**89.5**	60.5	2.6
IndexError	26	**41.0**	35.9	3.8	**50.0**	46.2	26.9
Other	105	**76.8**	47.9	15.6	**80.0**	46.7	13.3

		Crash class			Validation metric		
	n	MLE	DS	Coder	MLE	DS	Coder
Stage							
Draft	561	**78.2**	46.8	11.0	**74.0**	18.1	13.4
Improve	561	**79.1**	44.8	15.7	**85.7**	20.7	19.4
Debug	578	**72.7**	52.0	13.4	**67.5**	15.0	13.3
Outcome							
Clean run	525	–	–	–	**78.1**	18.6	16.3
Crash	1,094	**76.4**	48.2	13.2	–	–	–
Timeout / interrupt	81	–	–	–	–	–	–
Crash type							
ValueError	230	**67.4**	43.6	2.2	–	–	–
SyntaxError	161	**90.1**	34.8	12.0	–	–	–
TypeError	133	**71.2**	38.3	1.8	–	–	–
NameError	115	**96.5**	86.1	59.4	–	–	–
AttributeError	104	**75.6**	49.0	9.6	–	–	–
KeyError	86	**71.3**	57.0	13.2	–	–	–
RuntimeError	53	**61.0**	15.7	0.6	–	–	–
FileNotFoundError	43	**76.0**	65.1	17.1	–	–	–
ImportError	38	**89.5**	60.5	8.8	–	–	–
IndexError	26	39.7	**41.0**	3.8	–	–	–
Other	105	**76.8**	48.3	15.6	–	–	–

Table 11: **Judge validation on the held-out test set.** Majority agreement of the final judge prompt with the frozen human labels (%).

	Cases	Agreement
All cases	120	96.7
Actual failures	82	95.1
Crashes of the same class	40	93.3
Clean runs	20	100.0

J TEST-TIME SCALING ALGORITHMS

Each routine takes the current AIDE stage prompt q and budget N ($N = 4$ in Section 6.2) and returns one program. G generates code and $\mathrm{Pool}(q, N)$ independently draws $s_i \sim G(q)$ and predicts $\hat{y}_i = f_\theta(x_{s_i})$, using the full input of Section 4. For an ordered pool $\mathcal{P}$, $U(\mathcal{P})$ contains predicted usable programs and $M(\mathcal{P}) \subseteq U(\mathcal{P})$ those with finite predicted metrics. Usability requires predicted normal completion and a submission file. $d_\tau \in \{-1, +1\}$ orients the metric; ties retain generation order.

Algorithm 1 BoN: greedy selection by predicted metric

1: $\mathcal{P} \leftarrow \mathrm{Pool}(q, N)$
2: **if** $M(\mathcal{P}) \neq \varnothing$ and d_τ is known **then return** $\arg\max_{s \in M(\mathcal{P})} d_\tau \hat{m}(s)$
3: **if** $U(\mathcal{P}) \neq \varnothing$ **then return** its first program
4: **return** the least buggy program chosen by the failed pool judge from $\mathcal{P}$ (first on judge failure)

Algorithm 2 UCB: predicted quality plus an exploration bonus

1: $\mathcal{P} \leftarrow \mathrm{Pool}(q, N)$; $M \leftarrow M(\mathcal{P})$, excluding metrics the reviewer rejected
2: **if** $M \neq \varnothing$ and d_τ is known **then**
3: $\quad v(s) \leftarrow \big(d_\tau \hat{m}(s) - d_\tau m_{\mathrm{ref}}\big)/a_\tau$ for $s \in M$ ▷ BoN's score, same order
4: $\quad u(x_s) \leftarrow 1 - p_{\mathrm{clean}}(x_s)$; if any is missing, set all to zero ▷ $= 1 - p_{\max}(x_s)$ on M
5: $\quad$ **return** $\arg\max_{s \in M} \big[v(s) + \beta u(x_s)\big]$ ▷ $\beta = 0$: line 2 of Alg. 1
6: **return** the largest-$p_{\mathrm{clean}}(x_s)$ program in $U(\mathcal{P})$ (or $\mathcal{P}$ if empty; first if none is scored)

Lines 3–5 are Equation (5), with $\beta = 0.5$ at all stages in Table 2. The reference m_{ref} is the best verified real metric, else the parent's metric, else 0; the scale is $a_\tau = \max(|m_{\mathrm{ref}}|, 0.01)$, or the median $|\hat{m}|$ of the scored pool when no reference exists. Both are fixed for the pool and $a_\tau > 0$, so v is a positive affine map of BoN's $d_\tau \hat{m}$: the normalization changes the units in which the bonus is weighed, not the order or relative spacing of predicted metrics. A missing confidence disables the bonus for the whole pool.

Algorithm 3 Sequential: refine through all N predictions, then select

1: $q_1 \leftarrow q$; $\mathcal{P} \leftarrow [\,]$
2: **for** $i = 1, \ldots, N$ **do**
3: $\quad s_i \sim G(q_i)$; $\hat{y}_i \leftarrow f_\theta(x_{s_i})$; append $(s_i, \hat{y}_i)$ to $\mathcal{P}$
4: $\quad$ For the next draw, form q_{i+1} from $(s_i, \hat{y}_i)$: Improve if usable, otherwise Debug
5: **return** $\mathrm{Greedy}(\mathcal{P})$, applying lines 2–4 of Algorithm 1

Algorithm 4 Pairwise: single-elimination preference tournament

1: $\mathcal{P} \leftarrow \mathrm{Pool}(q, N)$; $A \leftarrow [1, \ldots, N]$
2: **while** $|A| > 1$ **do**
3: $\quad$ Compare adjacent pairs in A using the LLM judge, their code, and predicted reports
4: $\quad$ Advance each winner (left candidate on invalid verdict); carry an unpaired candidate
5: $\quad A \leftarrow$ the advanced candidates, preserving bracket order
6: **return** the remaining program ▷ $N - 1$ comparisons; three for $N = 4$

Algorithm 5 Fusion: synthesize a program from the predicted pool

1: $\mathcal{P} \leftarrow \mathrm{Pool}(q, N)$
2: $s_f \leftarrow G_{\mathrm{fuse}}(q, \mathcal{P})$, using candidate plans, code, and predicted reports
3: **if** synthesis returns a program **then return** s_f
4: **return** the first program in $U(\mathcal{P})$, or the first in $\mathcal{P}$ if none is usable

K UCB SELECTION ABLATIONS

Algorithms 1 (BoN) and 2 (UCB) share one ranking signal: both rank only candidates that the model predicts to run successfully, write a submission file, and report a finite metric, and both score them by the oriented predicted validation metric $d_\tau \hat{m}(s)$. UCB normalizes this score to a relative gain $v(s)$ with a reference shared across the pool and positive scale, which leaves BoN's ranking and relative gaps unchanged, and then adds the exploration bonus $\beta u(x_s)$; at $\beta = 0$ it picks BoN's candidate whenever a candidate ranked by metric exists. Two differences remain outside the bonus. UCB excludes metrics rejected by the reviewer, and when metric ranking is unavailable it chooses the candidate with the highest $p_{\text{clean}}(x_s)$ (using the full pool only when no candidate is usable), whereas BoN returns the first usable candidate or asks an LLM judge to choose from the failed pool if none is usable. The main BoN–UCB comparison therefore changes more than the exploration bonus. We report two controls within the UCB implementation: the Improve experiment varies only the bonus while holding Draft and Debug fixed; the Debug experiment holds Draft and Improve fixed and compares greedy selection by clean probability with metric UCB. Both controls evaluate complete searches. Appendix L separately examines the exploration weight across all three stages and its interaction with confidence gating.

Shared evaluation. Each comparison uses the same 17 tasks and three seeds matched between arms: seeds 11∼13 for the Improve comparison and 34∼36 for the Debug comparison, both at eight hours (51 trials per arm). The generator is Qwen3.5-9B, the world model is the 4B checkpoint used for these experiments (step 6,685), and feedback uses Qwen3-Coder-Next. Models, sampling, parent selection, and review settings are fixed within each pair. Leakage rejection is enabled; submission consistency review rejects in Improve and is advisory in Debug. Mean percentile averages seeds within each task and weights tasks equally; missing or invalid final submissions score zero.

Table 12: **UCB ablations by stage.** Each row contains all 51 completed trials. The Improve control uses the same metric eligibility, reviewer exclusions, and fallback as its UCB arm. The Debug comparison changes both the reward and eligibility. Compare rows within each block. Mean percentile ± SE across the three seed means.

Selection rule	Mean percentile	Valid submissions
Improve stage, eight hours; real check after each selection		
Metric greedy, $\beta = 0$	38.58 ±3.52	38/51
Metric UCB, $\beta = 1$	45.38 ±2.36	44/51
Debug stage, eight hours; real check after each selection		
Greedy, p_{clean}	41.88 ±2.10	48/51
UCB, $v + \beta u$, $\beta = 1$	48.37 ±1.82	48/51

K.1 IMPROVE STAGE BONUS WITH A SHARED GREEDY CONTROL

Intervention and controls. Draft and Debug each generate one candidate. Improve generates four and uses the same UCB code, metric scaling, predicted usability gate, reviewer exclusions, and largest-$p_{\text{clean}}(x_s)$ fallback in both arms. Only Improve's β changes, from 0 to 1 in $v(s) + \beta u(x_s)$. This is a greedy control within UCB, distinct from BoN in the main table. Here $p_{\max}(x_s)$ is the largest probability over clean and exception classes. Every eligible candidate is predicted to run cleanly, so $p_{\max}(x_s) = p_{\text{clean}}(x_s)$ there and the bonus uses the uncertainty score $u(x_s) = 1 - p_{\max}(x_s)$ of Equation (3).

Both arms receive eight hours and request a real check after every selected candidate, including selections other than the best, subject to the shared timeout skip and checks on the remaining budget. Both uncertainty sampling of candidate metrics and plan axis prompting are disabled. Leakage and submission consistency rejection are enabled. Both arms execute programs on four H100 GPUs with the same split of tasks. Model calls rotate over the same generator, world model, and review servers, so both arms use each server equally.

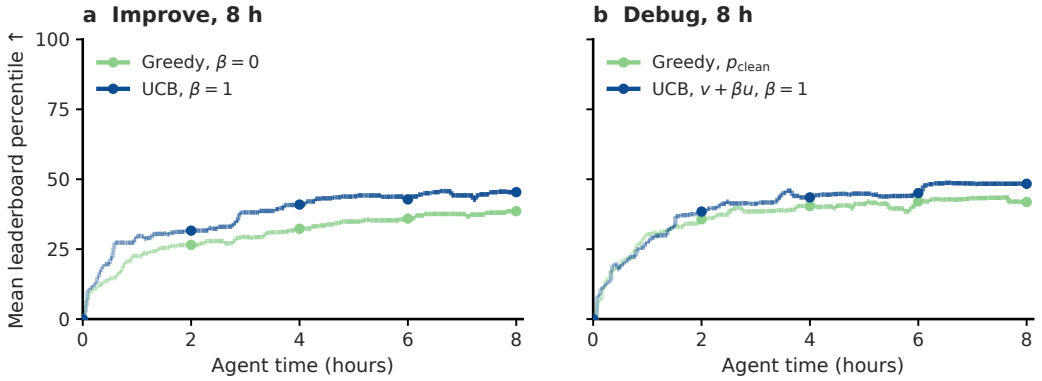


Figure 6: **Search trajectories for the ablations by stage.** (a) Improve uses $v(s) + \beta u(x_s)$ with $\beta = 0$ or 1, where $u(x_s) = 1 - p_{\max}(x_s) = 1 - p_{\text{clean}}(x_s)$ on eligible candidates; Draft and Debug each generate one candidate. (b) Debug compares Greedy (p_{clean}) with UCB ($v + \beta u$, $\beta = 1$); Draft and Improve use metric UCB ($v + \beta u$, $\beta = 1$) in both arms. Greedy is green and UCB blue in both panels. Curves credit real submissions after execution and review, carry the latest submission selected by the agent forward, and score zero before the first valid submission. All endpoints match the independent final grades. In the Improve comparison at eight hours, both arms use every model server equally; in the Debug comparison, both arms use H100 model servers. All plotted arms execute programs on H100 GPUs.

Results and search step counts. Greedy reaches 38.58 and UCB 45.38 mean percentile: +6.80 points, with 38/51 versus 44/51 valid submissions. Draft and Debug policies are fixed, but the resulting number of Improve decisions is not: the saved selection audit records 994 for Greedy and 1,249 for UCB over 51 trials each. This is a control of the selection rule under equal time budgets, not a replay with equal steps.

Table 12 and Figure 6(a) focus on the pair differing only in the Improve bonus.

K.2 Debug stage selector comparison

Intervention and controls. Every arm generates four candidates at each stage. Draft and Improve use the same $v(s) + \beta u(x_s)$ rule, with $\beta = 1$, throughout. Only Debug changes, between

$$\underbrace{p_{\text{clean}}(x_s)}_{\text{Greedy}}, \qquad \underbrace{v(s) + \beta u(x_s)}_{\text{UCB}}.$$

For probability greedy, a predicted candidate with a finite $p_{\text{clean}}(x_s) \in [0, 1]$ is eligible even when predicted to fail or lacking a usable metric; reviewer rejection of a predicted metric does not remove it from this probability ranking. Its fallback selects the highest known $p_{\text{clean}}(x_s)$, then the first usable candidate or first candidate if no probability is known. Ties retain generation order. Known unusable real results remain excluded.

The metric UCB reference retains Algorithm 2's eligibility rule for usable metrics and confidence fallback. Its comparison with probability greedy therefore changes both reward and eligibility, so it compares two selectors rather than isolating the exploration bonus. Each arm receives eight hours and requests a real check after every selected candidate, including selections other than the best, subject to the common timeout skip and remaining budget. Model servers and program execution use H100 GPUs throughout, and review calls are split evenly across arms.

Results. Greedy reaches 41.88 and UCB 48.37 mean percentile, with 48/51 valid submissions in each arm. UCB minus Greedy is +6.49 points, which is not established. This experiment does not test a failed pool judge.

L UCB EXPLORATION-WEIGHT SENSITIVITY

We compare $\beta \in \{0, 0.25, 0.5, 1, 2\}$ at three hours, then report completed eight-hour runs for $\beta = 0.5$ and 1, with and without the S50/F83 confidence gate. The UCB arms use p_{clean} at every stage, giving the selection score $v(s) + \beta(1 - p_{\text{clean}}(x_s))$; the same β applies to Draft, Debug, and Improve. Candidate width is four, with Qwen3.5-9B generation, the 4B world model at step 6,685, and Qwen3-Coder-Next feedback. Each setting contains the same 17 tasks and seeds 34∼36 (51 trials). Mean leaderboard percentile averages seeds within each task and weights tasks equally; missing or invalid submissions contribute zero.

Table 13: **Exploration weight and confidence gating.** Mean leaderboard percentile ± SE across the three seed means; every row retains all 51 trials. The $\beta = 0.5$ eight-hour rows are the UCB results in Table 2. Comparisons across experiment waves are descriptive.

Setting	β	Mean percentile	Valid submissions
(a) Ungated comparison at three hours			
Metric-only BoN reference†	0	36.47 ±0.10	38/51
p_{clean} UCB	0.25	37.28 ±4.87	38/51
p_{clean} UCB	0.5	41.71 ±2.89	38/51
p_{clean} UCB†	1	38.07 ±1.93	41/51
p_{clean} UCB	2	34.82 ±0.92	35/51
(b) Completed eight-hour runs			
Ungated p_{clean} UCB	0.5	45.11 ±1.06	45/51
Ungated p_{clean} UCB	1	46.09 ±2.45	47/51
S50/F83-gated p_{clean} UCB	0.5	49.23 ±1.02	46/51
S50/F83-gated p_{clean} UCB	1	48.39 ±1.56	47/51

†Three-hour checkpoints of eight-hour runs, retaining their eight-hour budget prompts. The $\beta = 0$ reference is metric-only BoN; its metric eligibility and fallback differ from UCB.

M GATE THRESHOLD SELECTION

The gate uses one pair of raw $p_{\max}$ thresholds at every stage: γ_{s} when MLE-World predicts success and γ_{f} when it predicts a failure other than a timeout. The two thresholds differ for two reasons. First, predicted successes and predicted failures have different $p_{\max}$ distributions, so they trade prediction error for skipped runtime differently (Figure 7), and a single threshold would place them at different operating points: about 15% error requires $\gamma_{\text{s}} = 0.90$ for successes but only $\gamma_{\text{f}} = 0.83$ for failures. Second, they differ in cost. Predicted successes, two-thirds of them Improve programs, are the long runs (median 115 s), so we want to skip as many of them as possible. Predicted failures, mostly Debug programs, usually crash within seconds (median 3 s), so running them is cheap and returns the exact error trace to repair. Predicted timeouts are never run (the timeout skip, Section 5.2): a real run of a program that times out occupies the executor for the full time limit and blocks the search, only to confirm a failure the prediction already reports.

Holdout tradeoff. We measure the tradeoff on MLE-World's 1,700 held-out programs, the MLE-World evaluation set of Section 6.1; it contains no programs from the agent runs and no test labels. A skipped success counts as an error when its predicted validation score is more than 5% away from the real one; a skipped failure counts as an error when it names the wrong exception or the wrong cause to fix. We pick two operating points per outcome: $\gamma_{\text{s}} \in \{0.90, 0.50\}$ (15.5%/17.8% error, 67.7%/99.6% of success runtime skipped) and $\gamma_{\text{f}} \in \{0.83, 0.56\}$ (15.0%/19.9% error, 55.9%/87.7% of failure runtime skipped).

Sweep over three pairs. Limited compute kept us from running all four combinations. The sweep instead tests the two moves we cared about: relaxing the success threshold to 0.50 to skip most long runs, then varying the failure threshold. We ran $(\gamma_{\text{s}}, \gamma_{\text{f}}) = (0.90, 0.83)$, $(0.50, 0.83)$, and $(0.50, 0.56)$ for three hours on the 17 tasks with a development seed (seed 37) that no reported run uses (Table 14). Every policy's best pair used $\gamma_{\text{s}} = 0.50$. The failure threshold split the policies:

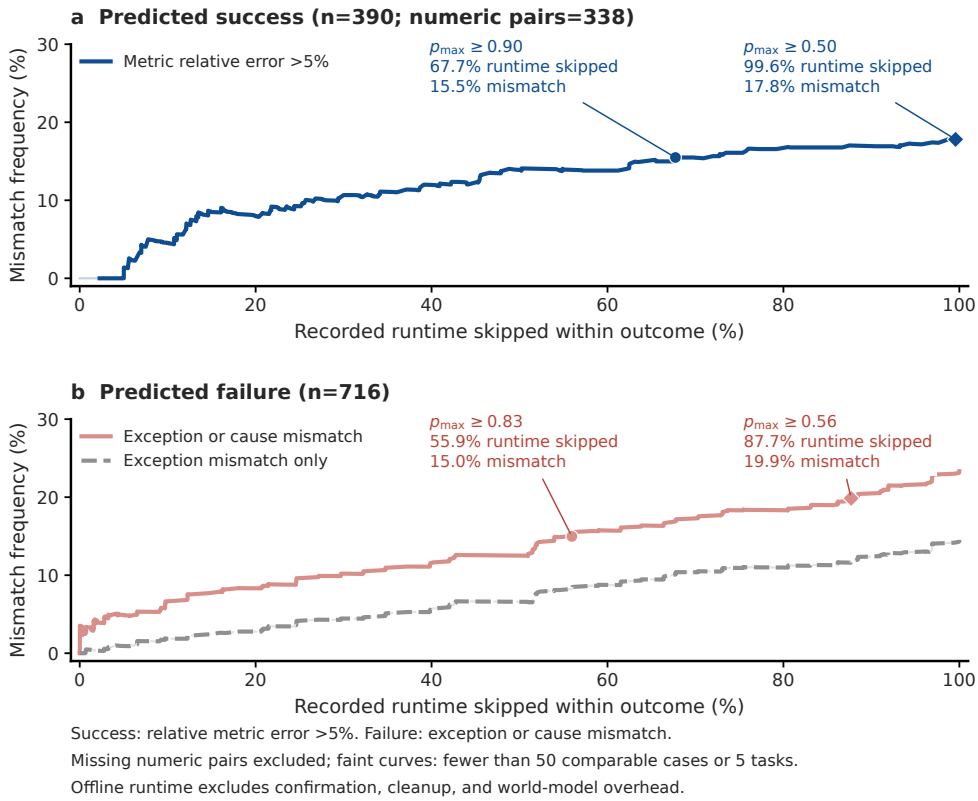


Figure 7: **Holdout prediction error versus skipped execution time.** Raw $p_{\max}$ gates on eligible Improve and Debug programs. Markers show the two success thresholds (0.90, 0.50) and two failure thresholds (0.83, 0.56). Success markers use the criterion of relative metric error exceeding 5%. Failure markers count exception or cause mismatches. Runtime percentages are computed separately within each predicted outcome. The held-out programs belong to the MLE-World evaluation of Section 6.1, not to any agent run. These estimates do not fully reflect the $p_{\max}$ distribution each policy produces during search; they propose candidate thresholds, and a separate development run chooses among them (Table 14).

0.56 was best for Sequential, Pairwise, and BoN, and 0.83 for Fusion and UCB. We froze each policy's best pair by its mean percentile on the development run and used it for the + *gate* runs of eight hours (seeds 34∼36) reported in Table 2.

Table 14: **Gate threshold sweep at three hours.** Each cell gives mean leaderboard percentile (mean completed outer iterations). Gated columns use (γ_s, γ_f) on the 17 tasks with development seed 37. The ungated column is the first three hours of the ungated runs of eight hours. Every policy's best pair uses $\gamma_s = 0.50$; $(0.90, 0.56)$ was not run.

	Mean percentile (mean iterations)			
		Gated (γ_s, γ_f)		
Policy	Ungated	$(0.90, 0.83)$	$(0.50, 0.83)$	$(0.50, 0.56)$
Sequential	32.31 (22.0)	37.39 (27.0)	20.86 (30.5)	**47.20** (37.6)
BoN	36.47 (29.6)	44.90 (49.1)	48.30 (69.4)	**52.25** (58.0)
Pairwise	40.44 (28.5)	40.19 (44.1)	38.60 (44.7)	**45.26** (71.6)
Fusion	34.01 (24.9)	38.90 (51.9)	**44.59** (44.6)	39.48 (45.4)
UCB ($\beta = 1$)	38.07 (32.9)	42.39 (37.2)	**49.19** (64.1)	42.26 (63.6)

N BoN random execution control

Question and control. For BoN, does confidence gating offer a useful tradeoff between quality and execution cost relative to simple random checks? We compare it with one fixed global probability $\rho = 0.50$, shared across tasks, seeds, stages, and predicted successes or failures. For each eligible selected program, the random gate requests a real check when a reproducible uniform draw is below ρ; numerical confidence does not affect that draw. Both gates retain predicted feedback, mandatory best/publication checks, fallbacks for missing confidence, the timeout skip, and budget checks. Thus $\rho = 0.50$ applies to discretionary decisions, not to all selected programs. Both BoN arms retain four candidates per step, the Qwen3.5-9B generator, and the same MLE-World checkpoint.

Evaluation. Each gate has 51 independently graded BoN trials on the same 17 tasks with a wall clock budget of eight hours. Random gating uses seeds 38∼40; confidence gating uses seeds 34∼36. We retain every trial, assign zero percentile to invalid submissions, average seeds within task, and weight tasks equally. Real execution time sums interpreter attempts and recovery cleanup once, including failures and retries; it excludes model inference and serving. All 51 random execution ledgers reconcile with their completed runs and final graded submissions.

BoN at matched counts of real execution. We focus on BoN because its usual selection path ranks predicted metrics without a comparative judge or synthesis. Both gates retain the failed pool judge and standard LLM review (Algorithm 1). Equal execution time permits different numbers of checks because durations vary. We therefore compare actual published submissions at the same *number of real executions* within each task. Each interpreter attempt counts once, including crashes, timeouts, mandatory checks, and retries in Draft, Debug, or Improve; cleanup adds no execution. For task t, set $B_t = \min_{g,s} N_{t,g,s}$, where N is the recorded count for gate g and seed s. This minimum over both gates and three seeds uses no test scores. We take the latest publication before attempt $B_t + 1$ begins, including review after attempt B_t, or zero before any valid publication. We preserve regressions in test score and all 17 tasks. The 102 complete BoN ledgers reconcile with 3,568 attempts and 623 publications; publication scores are unchanged and final endpoints match independent grades.

At these cutoffs, BoN with the confidence gate reaches mean percentile 47.24 versus 39.05 for random gating, a paired difference of $+8.19$ points. The mean cutoff is 15.1 executions, ranging from 2 to 32 across tasks. Stage composition, program identity, and duration can still differ. The cutoffs truncate finished searches of eight hours, so this compares the two gates after equal numbers of executions, not runs stopped at that count.

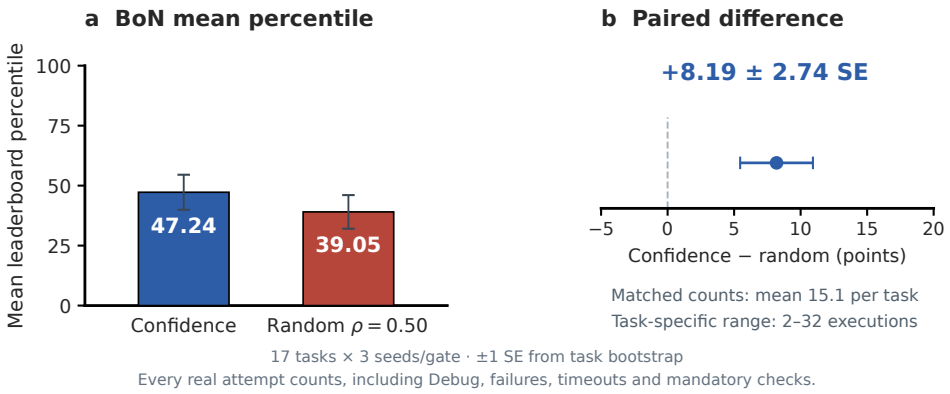


Figure 8: **BoN: confidence versus random gating at matched counts of real execution.** (a) Mean percentile. (b) Difference, confidence minus random. Whiskers: ± SE from 20,000 bootstrap resamples over tasks of seed means. Each gate has 17 tasks × three seeds. The cutoff for each task is the minimum attempt count across its six runs (mean 15.1; range 2∼32). Every real attempt counts, including Debug, failures, and retries. Existing trajectories are truncated retrospectively; stage proportions and execution durations can differ.

Table 15: **BoN: random execution versus confidence gating after eight hours.** Each gate has 51 trials. Percentile includes invalid submissions as zero; hours are real execution per trial. Δ is confidence minus random, paired by task, ± SE from 20,000 bootstrap resamples over tasks of seed means.

	Random $\rho = 0.50$			Confidence gate		Δ percentile
Policy	Valid	Pct.	Hours	Pct.	Hours	± SE
BoN	46/51	47.37	5.54	47.21	4.49	-0.15 ± 2.32

Endpoints at eight hours. For BoN, confidence gating uses 19.0% less real execution time than random gating: 4.49 versus 5.54 hours per trial, a paired difference of -1.05 hours. The percentile difference is -0.15 points, so no quality difference at eight hours is established (Table 15).

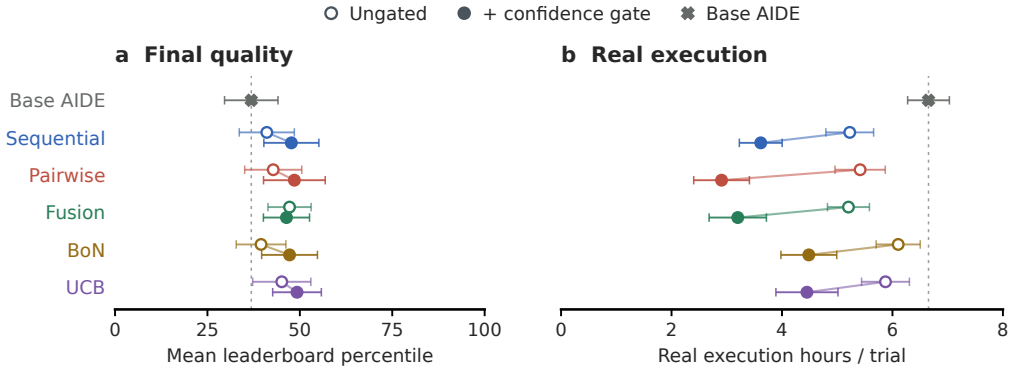


Figure 9: **Quality and execution cost with uncertainty.** Marginal ± SE from 20,000 bootstrap resamples over tasks of seed means for the eight-hour settings in Figure 1. Open circles: ungated; filled circles: confidence gate. Each setting has 17 tasks and three seeds. These endpoints at eight hours have unequal real execution costs; overlap of marginal SE bars does not test a paired difference (Table 2 gives paired differences from Base AIDE).

O AGENT EXECUTION STATISTICS

Capped real-BoN reference. The BoN (oracle) reference in Table 2 and Figure 1(c) runs four real candidates per step and selects by their observed validation scores. Each of the same 17 tasks and three seeds stops at 69 completed steps or a 16-hour cap: 18 trials reached the target and 33 hit the cap, averaging 51.20 steps. We independently grade the final published submissions; 48 are valid and the three missing submissions count as zero. Mean leaderboard percentile is 44.96 at 26.47 real-execution hours per trial, summing all 10,444 candidate attempts and cleanup. Parallel candidate durations add, so execution cost can exceed wall time.

Table 16 breaks down how the eight-hour policies of Table 2 spend their real executions. Without the gate, guided policies run fewer programs than Base AIDE, and a larger share of them finish cleanly. The gate removes 48∼73% of the remaining runs. Four of the five policies have a higher clean-run share with the gate; all five lose less time to timeouts, consistent with skipping programs whose outcome MLE-World predicts confidently. Split by agent stage, ungated guidance moves real execution from Debug to Improve. Per trial, Base AIDE spends 0.80 hours in Draft, 3.29 in Debug, and 2.56 in Improve (39% of its execution). Debug falls to 2.23, 2.49, 1.86, 2.37, and 2.23 hours for Sequential, Pairwise, Fusion, BoN, and UCB, and Improve takes 40%, 48%, 52%, 49%, and 48% of execution. Sequential is the exception: its Improve share stays near Base AIDE's. Clean Improve runs account for 1.19 hours in Base AIDE and 1.19, 1.61, 2.25, 2.01, and 1.84 hours in the same order. Gated policies record fewer verified new bests per trial, most visibly Fusion, but reach their first verified submission about as early. With the gate, policies also commit more programs to the search tree per trial (a guided step first predicts four candidates): 67.8 instead of 49.5 for Sequential, 103.5 instead of 65.2 for Pairwise, 77.8 instead of 59.8 for Fusion, 115.5 instead of 68.3 for BoN, and 111.9 instead of 71.9 for UCB. Base AIDE, which runs every program, commits 144.9.

Where the gain lands. Figure 10 compares, task by task, how much of Base AIDE's execution is lost to crashes and timeouts with what the guided policies gain there. Base AIDE scores worst on the tasks where it wastes most of its execution (a), and those are the tasks where guidance raises the rate of valid submissions (b): six of the nine more wasteful tasks are those on which Base AIDE sometimes ends without a valid submission. Percentile gains are larger on the more wasteful half of the tasks ($+10.8$ versus $+1.2$ points without the gate, $+15.8$ versus $+5.5$ with it), but their rank correlation with waste is not established (c); the two least wasteful tasks leave little headroom, with Base AIDE already at the 96th and 100th percentile.

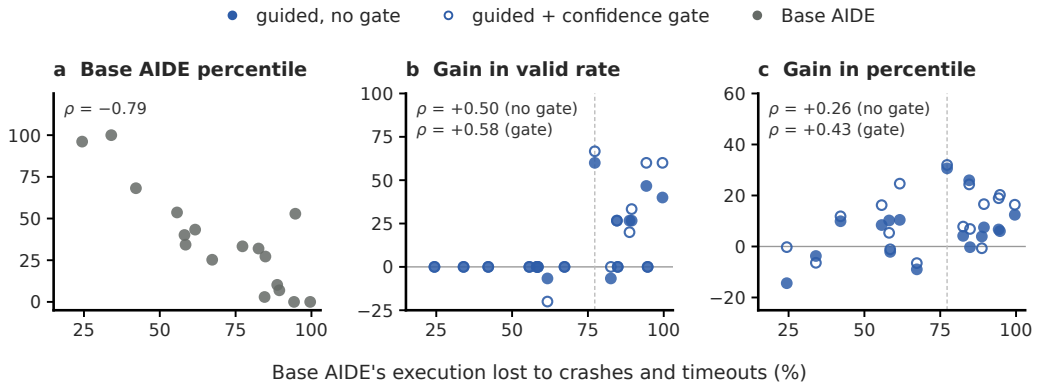


Figure 10: **Where the gain lands.** MLE-World's gains concentrate on the tasks where Base AIDE loses most of its execution to crashes and timeouts. Each point is one of the 17 tasks; x: share of Base AIDE's real execution time spent on runs that crash or time out (per trial, averaged over seeds). (a) Base AIDE's mean percentile. (b, c) Gain over Base AIDE in rate of valid submissions and mean percentile (points), averaged over the five guided policies, without (filled) and with (open) the confidence gate; the dashed line marks the median task. Spearman $\rho\pm$ bootstrap over tasks SE (20,000 resamples, omitting undefined correlations): (a) -0.79 ± 0.17; (b) 0.50 ± 0.18 without and 0.58 ± 0.16 with the gate; (c) 0.26 ± 0.25 and 0.43 ± 0.19.

Table 16: **Agent execution statistics.** Guidance raises the share of real runs that finish cleanly, and the confidence gate cuts real runs per trial by 48∼73%. The clean-run share rises with gating for four of five policies. 51 trials per row (17 tasks, three seeds, eight hours). Final submission: rate of valid submissions, mean leaderboard percentile as in Table 2, and mean percentile over valid submissions only. Real runs: executions per trial and their outcomes, pooled over all runs of a policy (clean: finished without an exception; timeout: hit the execution limit), and hours of real execution per trial, in total and in timed-out runs. Verified new bests: median hours to the first verified submission among trials that have one, the number of such trials, and verified new bests per trial.

	Final submission			Verified new bests		
Policy	Valid (%)	Mean pct.	Pct. if valid	First (h)	Trials with one	Per trial
Base AIDE	78.4	36.9	47.0	0.43	40	3.4
MLE-World guided						
Sequential	90.2	41.1	45.5	0.62	46	4.5
+ confidence gate	96.1	47.7	49.7	0.68	49	4.4
Pairwise	92.2	42.8	46.4	1.06	47	5.1
+ confidence gate	90.2	48.5	53.8	0.68	46	4.2
Fusion	96.1	47.2	49.1	0.56	49	16.5
+ confidence gate	94.1	46.4	49.3	0.75	48	7.1
BoN	88.2	39.5	44.8	0.93	45	6.9
+ confidence gate	94.1	47.2	50.2	0.58	48	6.3
UCB	88.2	45.1	51.1	0.57	45	11.3
+ confidence gate	90.2	49.2	54.6	0.39	46	5.9

	Real runs					
Policy	Per trial	Clean (%)	Crash (%)	Time-out (%)	Hours/ trial	Timeout h/trial
Base AIDE	144.9	13.4	83.8	2.8	6.65	3.33
MLE-World guided						
Sequential	47.8	34.1	60.0	5.9	5.23	2.25
+ confidence gate	21.6	52.6	39.6	7.8	3.62	1.33
Pairwise	63.9	28.5	67.3	4.2	5.41	2.04
+ confidence gate	17.1	56.8	35.3	7.9	2.91	0.98
Fusion	57.5	64.5	32.0	3.5	5.20	1.40
+ confidence gate	24.2	65.7	28.6	5.7	3.20	1.11
BoN	67.2	29.7	66.1	4.3	6.10	2.11
+ confidence gate	23.7	59.3	32.0	8.7	4.49	1.71
UCB	69.8	43.8	52.1	4.1	5.87	2.20
+ confidence gate	36.0	42.2	51.3	6.5	4.45	1.85

P DIAGNOSTICS ON FIXED STATES

Table 17 fixes the starting task state to isolate what MLE-World's predictions contribute to a local decision, separately from the rest of the search. We sample 153 states from recorded agent runs, three per task and stage (draft, debug, improve) on each of the 17 tasks, without consulting their outcomes. At each state, Qwen3.5-9B generates four candidates and MLE-World predicts their execution records. Selection, Pairwise, and Fusion use these same four candidates; Sequential shares the first candidate and generates three revisions under each feedback condition. We execute and grade every candidate and every program a rule produces; each rule's choice is recorded before any private test grade is read. We omit UCB because these local outcomes do not measure the value of exploration through later search decisions.

Screening and Fusion improve immediate outcomes. To isolate the contribution of predictions, we fix the task state and its four generated candidates, execute and grade every candidate, and vary

Table 17: **Diagnostics on fixed states.** Screening and Fusion improve validity and test percentile; the test percentile gains of Pairwise and Sequential from predicted reports are not established. Selection, Pairwise, and Fusion share their input pool; Sequential shares its starting candidate. *Uniform*: expected outcome of a random pick. *Screen*: random pick among candidates MLE-World predicts usable with a finite metric, else the largest $p_{\text{clean}}(x_s)$. + *ranking*: the screened candidate with the best predicted metric. *Real validation*: best real validation score, else the ranking's pick. Sequential uses the same selector by predicted metric and fallback in every condition; *Structured predictions* omit terminal output from revision feedback. Test pct. is zero if invalid. Δ: paired difference from the row above within each panel $\pm$ SE over 4,000 hierarchical task/run bootstrap resamples. †Executes all four inputs. ‡Uses real execution feedback during revision.

Rule	Valid (%)	Δ valid (points)	Test pct.	Δ test pct. (points)
Selection: pick one of the same four candidates (153 states)				
Uniform over the pool	26.1	–	10.9	–
Screen by predicted outcome	41.9	+15.8 ±3.1	16.4	+5.51 ±1.18
+ ranking by predicted metric	42.5	+0.5 ±1.3	17.1	+0.68 ±0.79
Real validation of all four†	47.1	+4.6 ±2.6	19.8	+2.65 ±2.19
Pairwise: judge the same four candidates (153 states)				
No execution report	32.7	–	13.0	–
Predicted reports	37.9	+5.2 ±3.6	15.6	+2.60 ±1.89
Real reports†	34.6	−3.3 ±2.2	14.6	−1.03 ±1.46
Sequential: refine the same starting candidate (150 states)				
No revision report	28.0	–	10.4	–
Structured predictions	31.2	+3.2 ±4.5	12.3	+1.86 ±2.16
Predicted reports	38.0	+6.8 ±4.3	15.3	+2.99 ±2.50
Real reports‡	39.4	+1.4 ±4.9	15.0	−0.26 ±3.48
Fusion: synthesize from the same four candidates (150 states)				
No execution report	36.7	–	14.0	–
Predicted reports	49.2	+12.5 ±4.5	20.6	+6.55 ±2.87
Real reports†	46.5	−2.7 ±4.3	20.4	−0.21 ±1.71

only how the predictions are used (Table 17). Screening and Fusion show the clearest gains in this controlled comparison. First, the predictions prevent failed runs: choosing among the candidates MLE-World predicts will run raises the share of chosen programs that yield a valid submission from 26.1% to 41.9% (+15.8 points), and predicted reports raise fusion's share from 36.7% to 49.2% (+12.5 points). As in complete runs, the chosen program executes about as long (256 versus 260 seconds), but more of that time goes to runs that finish cleanly (135 versus 96 seconds; +39 seconds). Second, test percentile rises by 5.51 and 6.55 points, mostly through this higher validity. Fusion scores about the same with predicted and real reports (20.6 versus 20.4 percentile).

Selection. Each rule picks one of the same four candidates. The screen picks uniformly among candidates predicted usable with a finite predicted metric and, when none qualifies, the candidate with the largest $p_{\text{clean}}(x_s)$. No candidate qualifies in 80 of the 153 states. In the 73 states where the screen keeps at least one candidate, it gains 9.16 percentile points; the fallback states gain 2.29 points. Ranking the screened candidates by predicted metric adds little. Selecting by the real validation scores of all four, which requires executing every candidate, reaches 19.8, 2.65 points above ranking by predicted metric.

Pairwise selection. The same four candidates enter a tournament whose judge receives no candidate execution report, predicted reports, or real reports. Predicted reports raise test percentile by 2.60 points over no report, which is not established; validity rises from 32.7% to 37.9%.

Sequential refinement. Each condition starts with the same first candidate and generates three revisions, using no revision report, structured predictions without terminal output, full predicted

reports, or real reports. All conditions select among their four attempts by predicted metric with the same fallback rule, so the intervention changes revision feedback rather than final selection. Full predicted reports raise test percentile by 4.85 points over no report and by 2.99 over structured predictions; neither gain is established.

Fusion. Fusion combines the same four candidates into a new program given no execution report, MLE-World's predicted reports, or the real reports from executing all four. Predicted reports raise the rate of valid submissions by 12.5 points and test percentile by 6.55, and real reports score no higher (-0.21 points). The comparison tests the full report; it does not isolate terminal output text from structured fields. Three states are excluded from every Sequential and Fusion condition because their runs did not complete (one exceeded the generator's context limit and two were stopped during execution); the exclusion uses completion status alone. Letting the three missing paired differences take any value from -100 to $+100$ bounds Fusion's gain from predicted reports between $+4.52$ and $+8.44$ points.

Connection to complete searches. On the 150 states shared by all four local methods, Fusion has the highest mean percentile (20.6), followed by greedy selection by metric (17.3), Pairwise (15.4), and Sequential (15.3). Fusion exceeds Pairwise by 5.22 points and Sequential by 5.31; its lead over greedy selection by metric ($+3.28$ points) is not established. Fusion's local result is consistent with its highest ungated mean in Table 2 (47.2), but the remaining policies do not retain their ordering. The diagnostic compares immediate outputs, with different costs of model calls; it does not measure incumbent quality at eight hours or gating. Its greedy rule by metric also uses the shared $p_{\text{clean}}(x_s)$ fallback, whereas BoN uses the first usable candidate or the failed pool judge (Algorithm 1). It therefore shows a local advantage for Fusion, not the policy ranking over complete searches.

Q CHOICE OF PRIMARY AGENT METRIC

MLE-bench compares each final submission with the competition's human leaderboard. Its medals use Kaggle's thresholds: gold requires the top 10% of teams on leaderboards with fewer than 100 teams, the top 10 on leaderboards with 100$\sim$249 teams, and the top $10+0.2\%$ otherwise. Ten of our 17 leaderboards have more than 1,000 teams, where gold means roughly the top 1%. Leaderboard percentile instead measures the share of entries a submission beats. We use percentile as the primary metric because medal rates are all or nothing: 12 of our 17 tasks never yield a gold, while one task with a small leaderboard yields more than half of all golds. Hambardzumyan et al. (2026) likewise rank agents by percentile because medal thresholds amplify noise and hide progress on hard tasks, and Foster et al. (2026) report a continuous normalized score. Tables 18 and 19 break down the 306 final submissions of the six eight-hour ungated policies in Table 2 by task and by policy.

One task supplies more than half of all golds. Detecting-insults has a leaderboard of 50 teams, so gold means the top five. Seventeen of its 18 submissions win gold; only one Fusion seed misses. The task accounts for 17 of the 31 golds and adds up to three golds, or 5.9 points of gold rate, to every policy without separating them.

Most tasks never produce a medal. Twelve tasks yield no gold in any submission and nine yield no medal of any kind, so medal rates ignore them. Percentile still separates policies on these tasks. Means for each policy range from 0.0 to 36.6 on mlsp-2013-birds, where three policies never produce a valid submission, and from 2.9 to 53.9 on English text normalization, where a single submission earns a medal.

Where golds occur, a single seed decides them. On the other four gold tasks, gold requires the 98.7th to 99.1st percentile or the best leaderboard score. On aerial-cactus, 150 teams share a perfect AUC of 1.0, so gold requires exactly 1.0: a submission scoring 0.99996 ranks at the 87.7th percentile and wins no medal. With three seeds per task, one seed moves a policy's gold rate by 2.0 points.

Gold rate can reverse the ranking. Base AIDE has twice Fusion's gold rate (11.8% versus 5.9%; paired difference -5.9 points), yet Fusion submits valid solutions more often (96.1% versus 78.4%) and reaches a higher mean percentile ($+10.4$ points). Pairwise exceeds Fusion in gold rate ($+9.8$

Table 18: **Medals by task.** More than half of all golds come from one task and twelve tasks yield none, yet policies still differ in percentile on tasks without medals. The 306 final submissions of the six eight-hour ungated policies in Table 2 (18 per task: six policies, three seeds). Teams: leaderboard size. Gold rank: the rank a submission must reach for gold. Gold pct.: percentile of a submission scoring exactly the gold threshold. Golds and Any medal count submissions. Policy pct.: lowest and highest of the six mean percentiles of the policies. †Gold requires the best leaderboard score, which many teams share: AUC 1.0 on aerial-cactus (150 teams) and log loss 0.0 on leaf-classification (24 teams).

Task	Teams	Gold rank	Gold pct.	Golds	Any medal	Policy pct. (min–max)
Tasks with a gold						
`detecting-insults-in-social-commentary`	50	5	90.0	17	17	81.3–100.0
`dogs-vs-cats-redux-kernels-edition`	1,315	12	99.1	4	9	33.3–95.5
`plant-pathology-2020-fgvc7`	1,318	12	99.1	4	5	10.1–92.9
`tabular-playground-series-dec-2021`	1,189	12	98.7	4	4	46.3–72.7
`aerial-cactus-identification`	1,221	12	87.7†	2	2	50.4–69.8
Medals, but no gold						
`nomad2018-predict-transparent-conductors`	879	11	98.7	0	15	62.7–96.1
`random-acts-of-pizza`	462	10	97.8	0	14	68.2–87.7
`text-normalization-challenge-english-language`	261	10	96.2	0	1	2.9–53.9
No medal						
`spooky-author-identification`	1,242	12	99.0	0	0	40.1–55.4
`leaf-classification`	1,596	13	98.5†	0	0	17.8–47.9
`dog-breed-identification`	1,281	12	99.1	0	0	17.6–42.8
`tabular-playground-series-may-2022`	1,152	12	98.4	0	0	27.5–36.6
`mlsp-2013-birds`	81	8	90.1	0	0	0.0–36.6
`new-york-city-taxi-fare-prediction`	1,485	12	99.2	0	0	5.3–25.3
`denoising-dirty-documents`	162	10	93.8	0	0	0.0–22.2
`jigsaw-toxic-comment-classification-challenge`	4,539	19	99.6	0	0	7.0–20.8
`text-normalization-challenge-russian-language`	163	10	93.9	0	0	9.8–19.2
All 17 tasks				31	67	

points) but trails it in mean percentile (-4.4 points). Table 19 shows why. Fusion's three golds contribute 5.9 of its 47.2 percentile points, and its 46 other valid submissions, at a mean 45.8th percentile, contribute 41.3. Pairwise's eight golds contribute 15.4 of its 42.8 points, but its 39 other valid submissions average only the 35.8th percentile, and its four invalid submissions score zero.

Percentile has its own limit. Where leaderboard scores cluster, small score gains move percentile sharply: on English text normalization, accuracy 0.964 ranks at the 6th percentile and 0.990 at the 47th. Averaging over 17 equally weighted tasks limits the influence of any one leaderboard. We therefore base conclusions on mean percentile and report medal rates only for comparison with MLE-bench.

Table 19: **Medals and percentile by policy.** Gold counts turn on a few submissions, whereas most of each policy's mean percentile comes from submissions without a gold. Counts over 51 trials per policy. Mean percentile (Total, as in Table 2) splits into points from gold medal trials and from other valid trials; invalid submissions contribute zero, and rounded components may not sum to the total.

	Golds				Mean percentile		
Policy	detecting-insults	Other	Any medal	Invalid	Gold trials	Other valid trials	Total
Base AIDE	3	3	11	11	11.8	25.1	36.9
Sequential	3	3	11	5	11.5	29.5	41.1
Pairwise	3	5	14	4	15.4	27.4	42.8
Fusion	2	1	11	2	5.9	41.3	47.2
BoN	3	0	9	6	5.9	33.6	39.5
UCB	3	2	11	6	9.8	35.3	45.1

Table 20: **Exploratory prediction on three tasks excluded from fine-tuning.** Accuracy (%, ± program-bootstrap SE) on 300 programs, 100 per task with balanced agent stages. The fields and scoring match Table 1; DeepSeek uses default reasoning effort here. Bold denotes the best value in each column.

Method	Terminal output	Exception class	Crash class	Validation metric
DeepSeek	42.1±2.8	**63.7**±2.8	45.4±3.2	21.3±5.8
Coder	21.1±2.2	30.3±2.6	18.9±2.4	31.9±6.8
MLE-World (4B, no CoT)	**53.3**±2.8	60.3±2.8	**56.3**±3.1	**44.7**±7.3

Table 21: **Shared operations that could support transfer.** The middle column names tasks present in fine-tuning. These are proposed mechanisms, not measured causal effects.

Held-out task	Related training tasks	Shared structure and remaining shift
APTOS Blindness Detection	Plant Pathology 2020; Dog Breed	Image classification with pretrained visual encoders; retinal severity introduces ordinal labels and a new metric.
3D Object Detection	Dog Breed; Aerial Cactus	Shared image, array, and PyTorch components can expose familiar import and file errors; 3D geometry and detection remain new task-specific computations.
RANZCR Catheter/Line	Plant Pathology 2020	Image preprocessing and visual prediction heads are shared; chest radiographs and multiple catheter/line labels change the target.

R EXECUTION PREDICTION ON TASKS EXCLUDED FROM FINE-TUNING

Scope and selection. We explore out-of-distribution prediction with respect to the tasks in the world model's fine-tuning set. We report three selected tasks from an earlier terminal-output ranking of 56 tasks absent from fine-tuning: APTOS Blindness Detection, 3D Object Detection, and RANZCR Catheter/Line Classification. We use a frozen sample of 100 programs from each task's existing 200-program confirmation set, uniformly without replacement within stage: 34 Draft, 33 Improve, and 33 Debug programs (seed 20260926). The same 300 programs are used for all methods. A fresh audit of all 497,056 fine-tuning examples finds neither these task identities nor an exact, whitespace-normalized copy of any evaluation program. APTOS and RANZCR occur in the broader execution archive but were excluded from fine-tuning; 3D Object Detection is also outside that archive's 29-task pool.

Why transfer may be possible. The task's subject matter can change while parts of its execution pipeline remain familiar. For example, retinal disease and plant disease classifiers both decode images, transform tensors, and fit prediction heads; errors in these operations can have the same causes and repair requirements. Table 21 relates the three tasks shown in Table 22 to fine-tuning tasks through these shared operations. This explanation is supported by code overlap at the library level: 89 of the 100 APTOS programs import PyTorch and 81 import TorchVision, compared with 98 and 97, respectively, in a fixed random sample of 100 Plant Pathology 2020 training programs. These observations motivate a transfer hypothesis; they do not establish that a particular training task caused a successful prediction. Removing related task families from fine-tuning, compared with removing the same number of randomly chosen training examples, would provide a direct test of that hypothesis.

Interpretation and future work. MLE-World has the highest terminal agreement on each of the three selected tasks. Pooled exception-class accuracy is 60.3% for MLE-World and 63.7% for DeepSeek, while their crash-class agreement is 56.3% and 45.4%, respectively. Table 22 reports per-task results. MLE-World's terminal agreement is 55.5% on the 235 program errors, 60.5% on the 49 clean executions, and 0% on the 16 timeouts. The sample is dominated by failed executions, so this experiment primarily tests whether familiar execution failures remain predictable on new tasks. The results support the possibility of useful transfer by a 4B world model without chain-of-thought. Learning a world model that generalizes reliably across task families, software environments, and successful training runs remains future work. A larger, prospectively selected task-held-out evaluation and training-family ablations would separate broad generalization from transfer through shared code components.

Table 22: **Per-task prediction accuracy on the three selected tasks.** Accuracy (%, ± program-bootstrap SE) using the same methods and metrics as Table 20. Each task has 100 programs: 34 Draft, 33 Improve, and 33 Debug. Gray † cells have fewer than ten metric references; SE is omitted when the metric is undefined in any bootstrap resample.

Method	Terminal output	Exception class	Crash class	Validation metric
APTOS Blindness Detection				
DeepSeek	49.3±4.9	**68.0**±4.5	**59.0**±5.6	16.7±8.6
Coder	18.3±3.8	32.0±4.6	15.3±4.2	16.7±9.1
MLE-World (4B, no CoT)	**53.0**±4.9	60.0±4.8	53.6±5.6	**50.0**±12.1
3D Object Detection				
DeepSeek	35.7±4.7	52.0±4.9	26.5±5.0	31.8±10.1
Coder	30.0±4.6	30.0±4.6	25.6±5.1	45.5±10.7
MLE-World (4B, no CoT)	**48.7**±5.1	**53.0**±5.1	**47.0**±5.8	**50.0**±10.9
RANZCR Catheter/Line				
DeepSeek	41.3±4.9	**71.0**±4.4	51.0±5.4	0.0†
Coder	15.0±3.5	29.0±4.6	15.7±4.0	**28.6**†
MLE-World (4B, no CoT)	**58.3**±4.7	68.0±4.6	**67.5**±5.0	14.3†