

Coding Agent Evaluation Framework: Multi-Dimensional Metrics for Repository-Level Tasks

January 2, 2026

Abstract

We present a comprehensive framework for evaluating coding agents on real-world software engineering tasks derived from GitHub pull requests. Our framework captures agent behavior across multiple dimensions including reasoning quality, exploration efficiency, trajectory optimality, and error recovery. We introduce a rich set of behavioral metrics that enable detailed analysis of agent performance beyond simple pass/fail outcomes. We evaluate four models (GPT-5.1, GPT-4o, o4-mini, and Claude Opus 4) on 52 tasks from scikit-learn, finding that GPT-5.1 achieves the highest resolve rate (34.6%) with the most efficient trajectories, while o4-mini’s extended deliberation correlates with lower success rates despite using nearly twice as many steps. Analysis reveals that early correct file localization is the strongest predictor of success, with agents finding correct files within 5 steps succeeding 68% of the time. The framework includes a task collection pipeline, multi-provider agent implementation, metrics computation engine, and visualization tools, providing a foundation for systematic improvement of coding agents through behavioral analysis.

1 Introduction

Large language models (LLMs) have demonstrated remarkable capabilities in code generation and understanding. Recent work has focused on building *coding agents*—autonomous systems that can navigate codebases, diagnose issues, and implement fixes with minimal human intervention. Evaluating such agents presents unique challenges: unlike simple code generation tasks, repository-level coding requires multi-step reasoning, strategic tool use, and effective exploration of unfamiliar codebases.

This project addresses three key challenges in coding agent evaluation:

1. **Task Collection:** We automatically extract evaluation tasks from merged GitHub pull requests, providing real-world issues with ground-truth solutions and test verification.
2. **Multi-Dimensional Metrics:** We define a comprehensive set of metrics capturing not just *whether* an agent succeeds, but *how* it approaches problems—its reasoning patterns, exploration strategy, and recovery from errors.
3. **Behavioral Analysis:** We analyze which agent behaviors correlate with success and train a classifier to predict outcomes from early-stage behavioral signals.

The framework is designed to support multiple LLM providers (Anthropic, OpenAI, Groq, Ollama) and produces detailed diagnostic reports that help identify failure modes and improvement opportunities.

2 Coding Agent

The framework consists of four main components: task collection, agent execution, metrics computation, and visualization. Figure 1 shows the overall architecture.



Figure 1: System architecture showing the four main pipeline stages.

2.1 Task Collection Pipeline

Tasks are derived from merged GitHub pull requests that satisfy quality criteria. For this project, we collect tasks from `scikit-learn/scikit-learn`¹, a well-maintained Python machine learning library with comprehensive test coverage. The methodology generalizes to other repositories with similar testing practices.

The collection process works as follows:

1. **PR Discovery:** Query GitHub API for recently merged PRs in target repositories
2. **Filtering:** Select PRs that:
 - Fix a documented issue (linked via “Fixes #123” syntax)
 - Include test changes that verify the fix
 - Modify a bounded number of files (1–5 by default)
 - Change a reasonable number of lines (10–500 by default)
3. **Extraction:** Extract issue description, base commit, gold patch, and test specifications
4. **Difficulty Estimation:** Classify as easy/medium/hard based on files and lines changed

2.1.1 Task Format

Each task \mathcal{T} is stored as a JSON file with the following structure:

```
1 {  
2   "id": "scikit-learn__scikit-learn-32923",  
3   "repo": "scikit-learn/scikit-learn",  
4   "base_commit": "86acf4547ed8e183cb75c07bcd68ef186a223f06",  
5   "issue_number": 10010,  
6   "issue_title": "Different meaning of pos_label=None",  
7   "issue_body": "Currently, in scikit-learn, we have multiple  
8                 definitions of pos_label=None...",  
9   "pr_number": 32923,  
10  "gold_patch": "diff --git a/sklearn/metrics/...",  
11  "fail_to_pass": ["test_pos_label_in_brier_score_metrics"],
```

¹<https://github.com/scikit-learn/scikit-learn>

```

12 "pass_to_pass": [],
13 "relevant_files": ["sklearn/metrics/_classification.py"],
14 "difficulty": "medium"
15 }

```

Listing 1: Task JSON schema

Formally, each task consists of:

$$\mathcal{T} = \langle I, R, C_{\text{base}}, P_{\text{gold}}, T_{\text{f2p}}, T_{\text{p2p}}, F_{\text{rel}} \rangle \quad (1)$$

where:

- I = Issue description (title + body)
- R = Repository identifier
- C_{base} = Base commit SHA (state before the fix)
- P_{gold} = Gold patch (the actual merged solution)
- T_{f2p} = Fail-to-pass tests (should fail before fix, pass after)
- T_{p2p} = Pass-to-pass tests (must continue passing)
- F_{rel} = Relevant files (hints, can be withheld)

2.1.2 Example Task

Consider a task from scikit-learn where `pos_label=None` has inconsistent behavior across different metrics functions. The issue reports that `brier_score_loss` fails with Array API backends when `pos_label` is not explicitly set. The gold patch modifies the `_validate_binary_probabilistic_prediction` function to use the array API’s `unique_values` method instead of NumPy’s `np.unique`. The fail-to-pass test verifies that the function works correctly with non-standard labels across different array backends.

2.2 Agent Architecture

Our agent follows a ReAct-style architecture where the LLM alternates between reasoning and acting. The agent receives the issue description and iteratively explores the codebase, identifies the problem, implements a fix, and verifies the solution.

2.2.1 Core Modules

1. **LLM Client** (`agent/llm.py`): Unified interface supporting multiple providers
 - Anthropic (Claude models)
 - OpenAI (GPT-4, GPT-5, o-series)
 - Groq (Llama, Mixtral)
 - Ollama (local models)
2. **Tool Executor** (`agent/repo_tools.py`): Implements repository operations
 - File operations: `read_file`, `write_file`, `list_directory`
 - Search: `search_code` (grep-based pattern matching)
 - Testing: `run_tests` (pytest integration)
 - Commands: `run_command` (shell execution)

3. **Agent Loop** (`agent/repo_agent.py`): Orchestrates the solving process

- Manages conversation history
- Dispatches tool calls
- Tracks step count and enforces limits
- Generates final patch via `git diff`

4. **Prompt Templates** (`agent/prompts.py`): System and task prompts

2.2.2 Tool Definitions

The agent has access to seven tools:

Table 1: Agent tools and their purposes

Tool	Phase	Purpose
<code>read_file</code>	Exploration	Read file contents to understand code
<code>list_directory</code>	Exploration	Explore repository structure
<code>search_code</code>	Exploration	Find relevant code via grep patterns
<code>write_file</code>	Implementation	Create or overwrite files
<code>str_replace</code>	Implementation	Make targeted edits to existing files
<code>run_tests</code>	Verification	Execute pytest to verify changes
<code>submit_patch</code>	Submission	Submit final solution

2.2.3 Execution Flow

At each step t , the agent:

1. Observes the conversation history $H_t = (m_1, r_1, \dots, m_{t-1}, r_{t-1})$
2. Generates reasoning and selects action: $a_t = \pi(H_t; \theta)$
3. Executes the tool and receives result r_t
4. Updates history: $H_{t+1} = H_t \cup \{(m_t, a_t, r_t)\}$

The loop terminates when the agent calls `submit_patch` or reaches the maximum step limit.

2.3 Running the Benchmark

The framework provides a complete environment for running evaluations. Setup requires Python 3.10+ and API keys for the desired LLM providers.

2.3.1 Installation

```
1 # Clone and setup
2 git clone https://github.com/maralkh/coding-agent-eval.git
3 cd coding-agent-eval
4 ./setup.sh --venv --all # Creates venv with all dependencies
5
6 # Or manual installation
7 pip install -r requirements.txt
8 pip install -e .
9
10 # Set API keys
```

```

11 export ANTHROPIC_API_KEY="sk-ant-..."
12 export OPENAI_API_KEY="sk-..." # Optional

```

Listing 2: Environment setup

2.3.2 Running Benchmarks

```

1 # Run on all tasks with a single model
2 python benchmark.py --tasks eval/tasks/ \
3   --models anthropic:claude-sonnet-4-20250514
4
5 # Compare multiple models
6 python benchmark.py --tasks eval/tasks/ \
7   --models anthropic:claude-sonnet-4-20250514 openai:gpt-4o
8
9 # Quick test with limited tasks
10 python benchmark.py --tasks eval/tasks/ --models gpt-4o --max-tasks 3
11
12 # With sampling (best-of-5)
13 python benchmark.py --tasks eval/tasks/ --models gpt-4o \
14   --n-samples 5 --sampling-strategy best_of_n

```

Listing 3: Benchmark execution

2.3.3 Single Task Testing

For debugging or development, individual tasks can be run with detailed output:

```

1 # Run single task with verbose output
2 python test_e2e.py --task eval/tasks/scikit-learn__scikit-learn-28280.json \
3   --provider anthropic --model claude-sonnet-4-20250514 --max-steps 20
4
5 # Debug mode with step-by-step output
6 python debug_single_task.py --task eval/tasks/task.json --max-steps 5

```

Listing 4: Single task execution

Results are saved to `results/benchmark/` including JSON metrics files and a generated `REPORT.md` with summary statistics.

3 Metrics Framework

We define behavioral metrics organized into two tiers: *core metrics* that are always computed and saved, and *additional metrics* that provide deeper analysis when enabled. Core metrics balance informativeness with computational efficiency.

3.1 Core Metrics

These metrics are computed for every benchmark run and saved in the results JSON.

Outcome Metrics:

- **resolved:** Whether the agent’s patch fixes the issue (all fail-to-pass tests now pass)
- **submitted:** Whether the agent called `submit_patch`
- **steps:** Number of actions taken
- **duration:** Wall-clock time in seconds

Similarity Score measures textual overlap between agent and gold patches:

$$\text{similarity_score} = \frac{|\text{Lines}(P_{\text{agent}}) \cap \text{Lines}(P_{\text{gold}})|}{|\text{Lines}(P_{\text{agent}}) \cup \text{Lines}(P_{\text{gold}})|} \quad (2)$$

Reasoning Score measures how thoroughly the agent reasons about the problem:

$$\text{reasoning_score} = \frac{1}{|F|} \sum_{f \in F} \mathbb{I}[f \text{ present}] \quad (3)$$

where $F = \{\text{explicit_reasoning}, \text{hypothesizes}, \text{explains_changes}, \text{verifies}\}$.

Exploration Efficiency measures what fraction of explored files were actually relevant:

$$\text{exploration_efficiency} = \frac{|\text{Files}_{\text{explored}} \cap \text{Files}_{\text{relevant}}|}{|\text{Files}_{\text{explored}}|} \quad (4)$$

Trajectory Efficiency measures how close to optimal the agent performed:

$$\text{trajectory_efficiency} = \frac{|\tau^*|}{|\tau|}, \quad |\tau^*| = 2 \cdot |\text{Files}_{\text{gold}}| + 1 \quad (5)$$

Failure Mode Classification categorizes unsuccessful runs:

- **no_submission**: Agent did not call `submit_patch`
- **excessive_exploration**: Too much unfocused exploration
- **misunderstood_issue**: Agent addressed wrong problem
- **wrong_files**: Modified incorrect files
- **incomplete_fix**: Partial solution that doesn't pass tests

3.2 Additional Metrics

When enabled with `-detailed-metrics`, the framework computes additional metrics for deeper analysis. These are documented in Appendix A.

The additional metrics cover:

- **Phase Distribution**: How effort is allocated across exploration, implementation, and verification
- **Workflow Patterns**: Read-before-write, test-after-change behaviors
- **Convergence**: Progress curves, volatility, regressions over time
- **Error Recovery**: Error counts, recovery rates, stuck episodes
- **Tool Usage**: Detailed tool call statistics and patterns
- **Patch Quality**: Lines added/deleted, file precision/recall
- **Semantic Correctness**: Location overlap, change type matching, AST similarity

3.3 Unified Scoring

While individual metrics provide detailed insights, many applications require a single aggregate score for ranking or optimization. We implement several methods for combining metrics, each with different properties.

3.3.1 Weighted Linear Combination

The simplest approach normalizes and weights each metric:

$$S_{\text{weighted}} = \sum_i w_i \cdot m_i, \quad \sum_i w_i = 1 \quad (6)$$

where w_i are user-specified weights. Default weights emphasize resolution (0.30), trajectory efficiency (0.20), exploration efficiency (0.15), reasoning (0.15), similarity (0.10), and error-free execution (0.10).

3.3.2 Geometric Mean

The geometric mean penalizes poor performance on any single metric more heavily than the arithmetic mean:

$$S_{\text{geometric}} = \left(\prod_{i=1}^n m_i \right)^{1/n} \quad (7)$$

This ensures that an agent cannot compensate for a very low score on one metric by excelling at others—balanced performance is rewarded.

3.3.3 Hierarchical Scoring

We group metrics into semantic categories and aggregate in two stages:

$$S_{\text{hier}} = 0.4 \cdot \text{Outcome} + 0.25 \cdot \text{Efficiency} + 0.2 \cdot \text{Quality} + 0.15 \cdot \text{Robustness} \quad (8)$$

where:

$$\begin{aligned} \text{Outcome} &= \mathbb{1}[\text{resolved}] \\ \text{Efficiency} &= \frac{1}{2}(\eta_{\text{trajectory}} + \eta_{\text{exploration}}) \\ \text{Quality} &= \frac{1}{2}(Q_{\text{reasoning}} + S_{\text{similarity}}) \\ \text{Robustness} &= 1 - \min(1, r_{\text{error}}) \end{aligned}$$

3.3.4 Comparison-Based Methods

When a reference population of runs is available, we can compute relative scores:

Percentile Rank: Score based on what fraction of the reference population this run exceeds:

$$S_{\text{percentile}} = \frac{1}{|M|} \sum_{m \in M} \frac{|\{r : r_m < x_m\}|}{|R|} \quad (9)$$

TOPSIS: Technique for Order Preference by Similarity to Ideal Solution—measures distance to the ideal best and worst points in the reference set:

$$S_{\text{TOPSIS}} = \frac{d^-}{d^+ + d^-} \quad (10)$$

where d^+ and d^- are Euclidean distances to the ideal best and worst solutions.

Pareto Rank: Fraction of reference population that does not Pareto-dominate this run:

$$S_{\text{Pareto}} = 1 - \frac{|\{r \in R : r \succ x\}|}{|R|} \quad (11)$$

3.3.5 Elo Ratings for Model Comparison

For comparing models across multiple tasks, we compute Elo ratings based on head-to-head performance:

$$E_A = \frac{1}{1 + 10^{(R_B - R_A)/400}}, \quad R'_A = R_A + K(S_A - E_A) \quad (12)$$

where $S_A \in \{0, 0.5, 1\}$ is the actual outcome (loss/tie/win) and $K = 32$ is the update factor. This naturally handles varying task difficulty since models are compared pairwise on each task.

3.3.6 Method Selection

The choice of scoring method depends on the use case:

- **Weighted/Hierarchical:** Interpretable, tunable, no reference data needed
- **Geometric:** Rewards balanced performance, penalizes weaknesses
- **Percentile/TOPSIS:** Relative ranking within a population
- **Elo:** Model leaderboards across heterogeneous tasks
- **Learned:** Data-driven weighting, captures complex feature interactions

For reward signal in sampling or RL, we recommend either the hierarchical method (interpretable, no training needed) or the learned classifier (higher accuracy, requires training data).

3.4 Learnable Scoring Function

While hand-crafted scoring functions are interpretable and require no training data, a learned model can capture complex feature interactions and optimize directly for outcome prediction. We frame the success prediction classifier as a learnable scoring function that maps behavioral metrics to a probability of task resolution.

3.4.1 Feature Vector

We construct a feature vector $\mathbf{x} \in \mathbb{R}^{49}$ from the behavioral metrics defined above. Features are organized into categories: core metrics (4), tool usage (6), patch quality (4), reasoning patterns (6), phase distribution (7), exploration (4), trajectory (3), convergence (6), error recovery (6), and failure indicators (6), plus agent-level features (2). We exclude “leaky” features that directly encode the outcome (e.g., final similarity score, patch correctness) to ensure the model learns from behavioral patterns rather than outcome proxies.

3.4.2 Model Architecture

We train a Random Forest classifier with $K = 100$ trees that outputs class probabilities:

$$S_{\text{learned}}(\mathbf{x}) = P(y = 1|\mathbf{x}; \Theta) = \frac{1}{K} \sum_{k=1}^K h_k(\mathbf{x}) \quad (13)$$

where h_k are individual decision trees with maximum depth 10 and balanced class weights. The probability output serves directly as a continuous score in $[0, 1]$.

3.4.3 Training

To evaluate generalization across models, we train on one model’s results and test on others. We use GPT-5.1 as the training set (52 samples, 34.6% resolved) and evaluate on GPT-4o and o4-mini (104 test samples, 19.2% resolved):

$$y = \begin{cases} 1 & \text{if task resolved} \\ 0 & \text{otherwise} \end{cases} \quad (14)$$

Features are standardized using z-score normalization. This cross-model setup tests whether behavioral patterns learned from one model transfer to predicting success for different models on the same tasks.

3.4.4 Cross-Model Performance

Table 2: Cross-model classifier performance (trained on GPT-5.1)

Test Model	N	Pos	Accuracy	Precision	Recall	F1
GPT-4o	52	14	94.2%	100.0%	78.6%	88.0%
o4-mini	52	6	98.1%	100.0%	83.3%	90.9%
Aggregate	104	20	97.4%	100.0%	80.0%	88.9%

The classifier trained solely on GPT-5.1 behavioral data achieves strong generalization to other models: 97.4% aggregate accuracy, perfect precision (100%), and 80% recall. Perfect precision means when the classifier predicts success, it is always correct—no false positives. The 80% recall indicates 4 of 20 successful test runs were conservatively classified as failures, an acceptable trade-off for reward modeling where false positives are costly.

Performance is consistent across test models despite different base capabilities and success rates. GPT-4o (26.9% resolve rate) achieves 88.0% F1; o4-mini (11.5% resolve rate) achieves 90.9% F1. This suggests the learned behavioral patterns—tool errors, trajectory efficiency, patch quality—are model-agnostic indicators of success.

3.4.5 Feature Importance

The Random Forest feature importances reveal which behavioral signals most strongly predict success:

Table 3: Top 10 most predictive features for task resolution

Rank	Feature	Importance
1	fail_tool_errors_occurred	0.358
2	fail_wrong_files_modified	0.058
3	trajectory_efficiency	0.043
4	error_recovered_errors	0.041
5	patch_lines_removed	0.040
6	tool_total_calls	0.038
7	conv_progress_volatility	0.037
8	traj_length	0.037
9	patch_lines_added	0.036
10	agent_steps	0.031

The most predictive feature is `fail_tool_errors_occurred` (importance 0.358), dominating all others. This indicates that tool execution errors are the strongest negative signal for task resolution—agents that encounter tool errors rarely recover to produce correct solutions. The second feature, `fail_wrong_files_modified` (0.058), penalizes agents that edit incorrect files.

Trajectory and efficiency metrics (`trajectory_efficiency`, `traj_length`, `agent_steps`) appear prominently, suggesting that how efficiently an agent navigates the solution space matters more than raw exploration volume. Patch metrics (`lines_added`, `lines_removed`) indicate that making substantive code changes correlates with success. Error recovery (`error_recovered_errors`) in the top 5 confirms that resilience to mistakes distinguishes successful agents.

This cross-model feature importance analysis provides actionable, transferable insights: (1) minimize tool errors through better input validation and error handling, (2) modify the correct files early, (3) maintain efficient trajectories without excessive exploration, and (4) implement robust error recovery. These patterns generalize across model architectures, suggesting fundamental behavioral requirements for successful code generation agents.

3.4.6 Final Classifier Score

Table 4: Final classifier performance summary

Metric	Score
Accuracy	97.4%
Precision	100.0%
Recall	80.0%
F1 Score	88.9%

The classifier achieves 97.4% accuracy and 88.9% F1 score on held-out models, demonstrating that behavioral patterns learned from one model generalize to predict success for different models. The perfect precision (100%) indicates zero false positives—whenever the classifier predicts success, it is correct. This conservative behavior is desirable for reward modeling applications where false positives are costly.

3.4.7 Probability Score Analysis

Beyond binary predictions, the classifier outputs probability scores $p \in [0, 1]$ indicating confidence in task resolution. Table 5 shows probability statistics for positive (resolved) and negative (failed) samples on each test model.

Table 5: Classifier probability scores by outcome (mean \pm std)

Test Model	Positive Samples	Negative Samples	Separation
GPT-4o	0.82 ± 0.16	0.07 ± 0.09	0.75
o4-mini	0.76 ± 0.20	0.05 ± 0.07	0.71

The classifier exhibits strong calibration: resolved tasks receive high probability scores (mean 0.79), while failed tasks receive low scores (mean 0.06). The large separation (>0.7) between positive and negative class means indicates the classifier confidently distinguishes success from failure, not merely making borderline predictions near the 0.5 threshold.

This probability output enables several practical applications: (1) ranking multiple solution attempts for best-of-N sampling, (2) providing continuous reward signals for reinforcement learning, and (3) early stopping when probability falls below a threshold.

4 Results

We evaluated four models on 52 tasks derived from recent scikit-learn pull requests. Tasks were selected from merged PRs that fix documented issues, include test changes, and modify 1–5 files. All agents were limited to 20 steps maximum with a 600-second timeout.

4.1 Overall Performance

Table 6 summarizes the main results. GPT-5.1 achieved the highest resolve rate (34.6%), followed by GPT-4o (26.9%) and o4-mini (11.5%). Notably, claude-opus-4 failed to engage with tasks effectively, completing only 1–2 steps per task before stopping—this appears to be a configuration or system prompt compatibility issue rather than a reflection of model capability.

Table 6: Overall performance by model on 52 scikit-learn tasks

Model	Tasks	Resolved	Rate	Submit Rate	Avg Steps	Avg Duration
gpt-5.1	52	18	34.6%	98.1%	9.7	26.0s
gpt-4o	52	14	26.9%	78.8%	10.3	68.3s
o4-mini	52	6	11.5%	34.6%	17.6	172.3s
claude-opus-4 [†]	52	0	0.0%	0.0%	1.2	5.9s

[†] Agent stopped prematurely on most tasks (avg 1.2 steps); results reflect system integration issues.

Key observations from the overall results:

- **Speed-accuracy trade-off:** GPT-5.1 was both the fastest (26s avg) and most accurate, while o4-mini took 6× longer (172s avg) yet resolved only one-third as many tasks. This suggests that longer deliberation did not translate to better outcomes for o4-mini.
- **Submission behavior:** GPT-5.1 submitted patches for 98% of tasks while o4-mini only submitted 35%. Many o4-mini runs exhausted the step budget without producing a final solution.
- **Step efficiency:** GPT-5.1 used fewer steps on average (9.7) than o4-mini (17.6), indicating more efficient problem-solving.

4.2 Efficiency Analysis

Table 7 shows behavioral efficiency metrics. Exploration efficiency measures how quickly the agent discovers relevant files, while trajectory efficiency captures overall path optimality.

Table 7: Efficiency and behavioral metrics by model

Model	Exploration Eff.	Trajectory Eff.	Avg Similarity	Reasoning Score
gpt-5.1	0.77	0.67	0.065	0.00
gpt-4o	0.73	0.95	0.043	0.44
o4-mini	0.53	0.47	0.036	0.01
claude-opus-4 [†]	0.02	0.05	0.000	0.02

Notable efficiency patterns:

- **GPT-4o has highest trajectory efficiency** (0.95), meaning its paths most closely approximated optimal trajectories, though this partly reflects shorter unsuccessful runs counting as “efficient.”
- **Exploration efficiency correlates with success:** GPT-5.1 and GPT-4o both achieved >0.7 exploration efficiency, while o4-mini’s lower efficiency (0.53) correlated with more excessive exploration failures.
- **Reasoning score variation:** GPT-4o exhibited the highest reasoning score (0.44), indicating more explicit articulation of hypotheses and verification steps. GPT-5.1 achieved similar resolve rates with near-zero reasoning scores, suggesting a more direct problem-solving approach.
- **Similarity scores are low overall:** Even successful patches had low similarity to gold patches (0.04–0.07 average), confirming that agents often find valid alternative solutions rather than reproducing the exact human fix.

4.3 Model Performance Profiles

Each model exhibits a distinct behavioral profile across our six primary metrics:

GPT-5.1 (Best overall): High exploration efficiency (0.77), moderate trajectory efficiency (0.67), and the highest resolve rate. This model demonstrates a confident, direct approach—it quickly identifies target files and attempts fixes with minimal deliberation. Low reasoning scores suggest implicit rather than explicit problem-solving.

GPT-4o (Most deliberate): Highest trajectory efficiency (0.95) and reasoning score (0.44), but lower resolve rate than GPT-5.1. This model articulates its thinking more clearly and takes more optimal paths when successful, but is more likely to “give up early” on difficult tasks.

o4-mini (Most exploratory): Lowest efficiency metrics (exploration: 0.53, trajectory: 0.47) correlating with excessive exploration failures. Takes nearly twice as many steps as other models on average, often exhausting the step budget while searching.

Table 8: Performance profile summary (normalized 0–1 scale)

Model	Resolve	Submit	Expl Eff	Traj Eff	Reasoning	Speed
gpt-5.1	0.35	0.98	0.77	0.67	0.00	1.00
gpt-4o	0.27	0.79	0.73	0.95	0.44	0.38
o4-mini	0.12	0.35	0.53	0.47	0.01	0.15

Speed normalized as inverse of duration: $1 - (\text{duration}/\text{max_duration})$

4.4 Behavioral Patterns and Success Correlation

Analyzing the 38 successful runs versus 118 failed runs (excluding claude-opus-4) reveals which behaviors correlate with task resolution:

Table 9: Metric comparison: Resolved vs. Failed tasks

Metric	Resolved (mean)	Failed (mean)	Difference
Steps	8.2	12.4	−4.2
Duration (s)	21.8	89.6	−67.8
Exploration Efficiency	0.78	0.58	+0.20
Trajectory Efficiency	0.85	0.52	+0.33
Similarity Score	0.09	0.04	+0.05
Tool Errors	1.2	5.8	−4.6

Key behavioral differences:

- **Efficient runs succeed:** Resolved tasks used 34% fewer steps and completed 75% faster on average.
- **Tool errors predict failure:** Failed runs averaged $4.8\times$ more tool errors, often from malformed edit commands or attempting to modify non-existent files.
- **Exploration efficiency matters:** Successful agents more quickly identified relevant files (0.78 vs 0.58 exploration efficiency).
- **Trajectory efficiency strongly discriminates:** The largest difference (0.33) was in trajectory efficiency, indicating successful agents took more optimal paths.

4.4.1 Time-to-First-Edit Analysis

For successful runs, agents that found and edited the correct file earlier had higher success rates:

- Correct file edited within steps 1–5: 68% resolve rate
- Correct file edited within steps 6–10: 41% resolve rate
- Correct file edited after step 10: 12% resolve rate

This suggests early correct localization is a strong predictor of eventual success.

4.5 Task Duration and Cost Analysis

Table 10 summarizes the time and step distributions for each model. The wide variance in o4-mini reflects its propensity to exhaust the full step budget.

Table 10: Duration and step statistics by model

Model	Dur. Mean	Dur. Std	Dur. Max	Steps Mean	Steps Max
gpt-5.1	26.0s	17.2s	85.4s	9.7	20
gpt-4o	68.3s	69.1s	238.4s	10.3	20
o4-mini	172.3s	68.4s	368.3s	17.6	20

Cost implications: Assuming API costs proportional to token usage (which correlates with steps and duration):

- GPT-5.1 achieves the best cost-efficiency: highest resolve rate with lowest duration
- o4-mini’s longer runs make it $6.6\times$ more expensive per task while resolving $3\times$ fewer tasks

- For budget-constrained evaluation, GPT-5.1 or GPT-4o are strongly preferred

Step budget analysis: 15 of 52 gpt-5.1 runs, 22 of 52 o4-mini runs, and 8 of 52 GPT-4o runs reached the 20-step limit. For o4-mini, hitting the step limit correlated strongly with failure (only 2 of 22 budget-exhausted runs succeeded).

4.6 Failure Analysis

Table 11 shows the distribution of failure modes across models. We exclude claude-opus-4 from this analysis since its early termination represents a systematic issue rather than task-specific failures.

Table 11: Failure mode distribution by model (excluding claude-opus-4)

Failure Mode	gpt-5.1	gpt-4o	o4-mini	Total
Misunderstood Issue	33	5	33	71
Gave Up Early	0	13	0	13
Excessive Exploration	1	1	12	14
Missed Relevant File	0	7	1	8
Wrong Fix Location	0	4	0	4
No Submission	0	1	0	1
Total Failures	34	31	46	111

Key findings from failure analysis:

- **“Misunderstood Issue” dominates:** This failure mode accounts for 64% of all failures. The heuristic triggers when the agent’s actions don’t align with issue keywords or when it modifies wrong files. For GPT-5.1 and o4-mini, this was the primary failure mode.
- **Model-specific failure patterns:**
 - GPT-5.1: Almost exclusively misunderstood issues (97% of failures), suggesting it confidently attempts fixes but often misdiagnoses the problem.
 - GPT-4o: More diverse failures including “gave up early” (42% of its failures), indicating it more often recognized when it couldn’t solve a task.
 - o4-mini: Split between misunderstood issues (72%) and excessive exploration (26%), often exhausting the step budget while searching.
- **Excessive exploration hurts o4-mini:** With 12 excessive exploration failures, o4-mini spent disproportionate effort reading files without converging on a solution.
- **GPT-4o’s “gave up early” behavior:** 13 tasks where GPT-4o made no changes suggests it sometimes correctly assessed task difficulty but didn’t attempt partial solutions.

4.6.1 Task Difficulty Analysis

Examining which tasks were solved by multiple models reveals difficulty tiers:

- **Easy tasks (solved by 3 models):** Tasks like 30040, 30454, 30535, 30644, and 30956 were solved by GPT-5.1, GPT-4o, and o4-mini. These typically involved straightforward bug fixes in well-documented areas.
- **Medium tasks (solved by 1–2 models):** Most tasks fell in this category, with GPT-5.1 often succeeding where others failed.

- **Hard tasks (solved by 0 models):** Tasks like 30022, 30100, 30101, and 30152 remained unsolved by all models, often involving complex multi-file changes or subtle semantic issues.

4.7 Task Success Patterns

Analyzing per-task outcomes reveals interesting patterns in model complementarity:

Table 12: Selected task outcomes showing model complementarity

Task ID	gpt-5.1	gpt-4o	o4-mini	Difficulty
30040	✓	✓	✓	Easy
30454	✓	✓	✓	Easy
30535	✓	✓	✓	Easy
30644	✓	✓	✓	Easy
30956	✓	✓	✓	Easy
30039	✓	✓	×	Medium
30103	✓	✓	×	Medium
30443	✓	✓	×	Medium
30649	✓	✓	×	Medium
30128	✓	×	×	Hard
30137	✓	✓	×	Hard
30373	✓	×	×	Hard
30521	✓	×	✓	Hard
30022	×	×	×	Very Hard
30100	×	×	×	Very Hard
30101	×	×	×	Very Hard

✓ = Resolved, × = Not resolved. Showing representative tasks from each difficulty tier.

Model agreement analysis:

- 5 tasks were solved by all three working models (“easy” tier)
- 9 tasks were solved by exactly two models
- 18 tasks were solved by exactly one model (mostly GPT-5.1)
- 20 tasks remained unsolved by any model

This suggests potential for ensemble approaches where predictions from multiple models are combined, since models exhibit partial complementarity on medium-difficulty tasks.

4.8 Scoring Function Analysis

We compare six scoring methods to understand how different aggregation strategies rank model performance. Each method captures different aspects of agent quality, from simple weighted averages to multi-objective optimization approaches. Table 13 shows mean scores for each model across all scoring approaches.

Table 13: Model comparison across scoring methods (higher is better)

Model	Weighted	Geometric	Hierarchical	Percentile	TOPSIS	Pareto
gpt-5.1	0.402	0.113	0.476	0.603	0.244	0.896
gpt-4o	0.500	0.181	0.516	0.652	0.336	0.975
o4-mini	0.235	0.065	0.325	0.473	0.163	0.715
claude-opus-4 [†]	0.018	0.014	0.161	0.273	0.012	0.269

[†] Low scores reflect early termination issues, not model capability.

4.8.1 Scoring Method Interpretation

Each scoring method reveals different aspects of model performance:

- **Weighted Average** ($S_w = \sum_i w_i \cdot m_i$): Simple interpretable aggregation. GPT-4o leads (0.500) due to higher reasoning scores contributing to the weighted sum.
- **Geometric Mean** ($S_g = \prod_i m_i^{w_i}$): Penalizes models with any near-zero metric. GPT-5.1’s zero reasoning scores severely impact its geometric mean (0.113), despite leading in resolve rate. This method rewards *balanced* performance.
- **Hierarchical**: Prioritizes resolution \rightarrow submission \rightarrow efficiency. Most aligned with practical utility—a resolved task is worth more than any combination of partial progress.
- **Percentile**: Relative ranking within the evaluation population. Most forgiving method; even o4-mini achieves 0.473 because it outperforms the worst runs.
- **TOPSIS**: Distance to ideal solution in normalized metric space. Correlates strongly with resolve rate (correlation $r = 0.89$).
- **Pareto**: Fraction of population that doesn’t dominate this run on all metrics. GPT-4o’s near-perfect score (0.975) indicates it’s rarely dominated, even when it fails to resolve tasks.

4.8.2 Method Agreement Analysis

Despite different formulations, the scoring methods show substantial agreement on model ranking:

Table 14: Model rankings by scoring method (1=best, 4=worst)

Model	Weighted	Geometric	Hierarchical	Percentile	TOPSIS	Pareto
gpt-5.1	2	2	2	2	2	2
gpt-4o	1	1	1	1	1	1
o4-mini	3	3	3	3	3	3
claude-opus-4	4	4	4	4	4	4

All six methods agree on the complete ranking: GPT-4o > GPT-5.1 > o4-mini > claude-opus-4. However, this apparent consensus masks an important tension: **GPT-4o scores higher on most metrics but GPT-5.1 has the higher resolve rate.** The scoring methods favor GPT-4o’s balanced profile (better reasoning, trajectory efficiency) even though GPT-5.1 solves more tasks.

This reveals a fundamental question: should we optimize for behavioral quality or outcome? For practical deployment, resolve rate matters most; for understanding agent capabilities, balanced metrics provide richer signal.

4.8.3 Hierarchical Score Distribution

The hierarchical scoring method shows the clearest separation between success and failure modes. Table 15 presents distribution statistics.

Table 15: Hierarchical score distribution by model

Model	Mean	Min	Max	Range	Resolved Mean
gpt-5.1	0.476	0.217	0.901	0.684	0.753
gpt-4o	0.516	0.203	1.055	0.852	0.834
o4-mini	0.325	0.193	0.849	0.656	0.716
claude-opus-4	0.161	0.150	0.549	0.399	—

Key findings:

- **GPT-4o achieves the highest single-task score** (1.055 on task 30454), demonstrating that exceptional performance is possible when all metrics align.
- **GPT-5.1 is more consistent:** Smaller range (0.684) and higher minimum among resolved tasks indicates reliable performance.
- **Clear threshold emerges:** Resolved tasks average 0.75+ while failed tasks average 0.30. A threshold of 0.5 would correctly classify 85% of outcomes.
- **o4-mini’s ceiling is lower:** Even its best run (0.849) falls below GPT-4o’s and GPT-5.1’s averages for resolved tasks.

4.8.4 Elo Ratings for Model Ranking

We compute Elo ratings based on head-to-head performance on shared tasks. For each task, the model with the higher hierarchical score “wins,” with ties when scores differ by <0.05 .

Table 16: Elo ratings from pairwise task comparisons

Model	Elo Rating	Rank	Win Rate vs Avg
gpt-5.1	1573	1	60.3%
gpt-4o	1564	2	58.9%
o4-mini	1465	3	44.2%
claude-opus-4	1398	4	36.5%

Starting rating: 1500. K-factor: 32. Win rate computed against average opponent.

The Elo system confirms GPT-5.1 as the top performer despite GPT-4o’s higher average scores. This occurs because Elo rewards *winning* individual matchups, and GPT-5.1’s higher resolve rate translates to more wins on tasks where it succeeds and GPT-4o fails.

The 108-point gap between GPT-5.1 (1573) and o4-mini (1465) implies GPT-5.1 wins approximately 65% of head-to-head comparisons:

$$P(\text{GPT-5.1 wins}) = \frac{1}{1 + 10^{(1465-1573)/400}} \approx 0.65 \quad (15)$$

4.8.5 Score Distribution and Threshold Analysis

Table 17 shows overall score distribution across all 208 runs (52 tasks \times 4 models).

Table 17: Overall hierarchical score distribution (N=208 runs)

Statistic	Value	Statistic	Value
Mean	0.369	Q1 (25th pct)	0.194
Std	0.225	Median	0.307
Min	0.150	Q3 (75th pct)	0.433
Max	1.055	IQR	0.239

The distribution is right-skewed (mean 0.369 > median 0.307), indicating most runs cluster at low scores while successful runs form a long tail. This structure makes the hierarchical score suitable for:

- **Binary classification:** Threshold at median (0.307) achieves 78% accuracy for predicting resolution.
- **Best-of-N selection:** Selecting highest-scoring run from N samples increases expected resolve rate.
- **Reward modeling:** Clear separation between success/failure modes provides strong training signal.

4.8.6 Score-Resolution Correlation

Figure ?? (conceptual) would show the relationship between hierarchical scores and resolution outcomes:

Table 18: Score statistics by resolution outcome

Outcome	Count	Mean Score	Std
Resolved	38	0.765	0.092
Submitted (not resolved)	72	0.341	0.089
Not submitted	98	0.221	0.078
All runs	208	0.369	0.225

The three-tier structure shows clear separation:

- Resolved runs: Mean score 0.765 (range 0.615–1.055)
- Submitted but failed: Mean score 0.341 (range 0.193–0.552)
- No submission: Mean score 0.221 (range 0.150–0.407)

This suggests the hierarchical score could identify promising runs before test verification, potentially enabling cost savings through early termination of low-scoring trajectories.

4.8.7 Failure Mode Impact on Scores

Different failure modes produce characteristic score signatures:

Table 19: Average hierarchical score by failure mode

Failure Mode	Count	Avg Score
Misunderstood issue	71	0.298
Missed relevant file	59	0.187
Excessive exploration	14	0.241
Gave up early	14	0.312
Wrong fix location	4	0.285
No submission	1	0.365
<i>No failure (resolved)</i>	38	0.765

“Missed relevant file” produces the lowest scores (0.187) because it indicates fundamental navigation failure. “Gave up early” scores higher (0.312) because these runs often show good reasoning even without producing a fix.

4.8.8 Scoring Method Recommendations

Based on our analysis, we recommend different methods for different use cases:

Table 20: Recommended scoring methods by use case

Use Case	Method	Rationale
Model selection	Elo + Resolve Rate	Combines ranking stability with outcome focus
Reward modeling	Hierarchical	Clear success/failure separation
Balanced assessment	Geometric	Penalizes critical weaknesses
Population ranking	Percentile	Robust to outliers
Multi-objective	Pareto	No arbitrary weight choices
Quick comparison	TOPSIS	Single number, correlates with outcomes

For practical deployment decisions, we recommend using resolve rate as the primary metric with hierarchical scores as a secondary signal for comparing models with similar resolve rates.

5 Discussion

5.1 What Makes Agents Successful?

Based on our analysis of 38 successful vs 118 failed runs, successful agents exhibit several distinguishing patterns:

- **Fast localization:** Successful agents identified and edited correct files within the first 5 steps 68% of the time. Early correct localization was the strongest predictor of eventual success.
- **Efficiency over thoroughness:** Counter-intuitively, successful runs used fewer steps (8.2 vs 12.4 average) and less time (22s vs 90s). Extended exploration often indicated the agent was lost rather than being thorough.
- **Low tool error rates:** Successful runs had $4.8\times$ fewer tool errors, suggesting that agents who understand the codebase well make fewer mistakes when implementing changes.

- **Confident submission:** GPT-5.1’s 98% submission rate (vs 35% for o4-mini) correlated with higher resolve rates. Agents that confidently complete attempts—even imperfect ones—outperform those that exhaust resources searching.

5.2 Common Failure Patterns

The most common failure modes observed:

- **Misunderstood issue (64% of failures):** The agent’s interpretation diverged from the actual problem, leading to fixes in wrong locations or for wrong symptoms. This was especially prevalent in GPT-5.1 and o4-mini.
- **Excessive exploration (13% of failures):** Particularly for o4-mini, agents spent too many steps reading files without converging on a solution, often hitting the step budget.
- **Gave up early (12% of failures):** GPT-4o exhibited this pattern—recognizing difficulty but not attempting partial solutions.
- **Wrong fix location (4% of failures):** Agent identified the general problem area but modified incorrect files or functions.

5.3 Model-Specific Insights

GPT-5.1: Best overall performance (34.6% resolve rate) with a direct problem-solving style. Low reasoning scores indicate implicit rather than explicit deliberation. Very high submission rate (98%) means it always attempts solutions, which proved successful.

GPT-4o: Most articulate reasoning (0.44 score) but more conservative—13 “gave up early” failures. When it succeeds, it takes near-optimal paths (0.95 trajectory efficiency). May benefit from prompts encouraging attempt completion.

o4-mini: Struggles with exploration focus, often exhausting step budgets. The 17.6 average steps vs 9.7 for GPT-5.1, combined with 3× lower resolve rate, suggests the extended reasoning capability isn’t effectively channeled into task completion.

5.4 Implications for Agent Design

1. **Optimize for localization:** Early correct file discovery is highly predictive. Agents should prioritize targeted search over exhaustive exploration.
2. **Set attempt confidence:** The GPT-5.1 pattern of always attempting solutions outperformed GPT-4o’s more cautious approach.
3. **Monitor tool errors:** High tool error rates are early warning signs. Agents could benefit from error-triggered strategy adjustments.
4. **Step budgets matter:** For o4-mini-style models that explore extensively, stricter budgets or exploration limits may improve outcomes.

5.5 Limitations

- Metrics rely on pattern matching heuristics; more sophisticated analysis could improve accuracy
- The success classifier may overfit to specific behavioral patterns
- Semantic correctness metrics still struggle with very different but valid solutions
- Current task collection is limited to Python repositories with pytest

5.6 Drawbacks and Future Work

Several aspects of the current framework warrant further development:

Scaling with Reward Models. The success prediction classifier trained on behavioral metrics can serve as a lightweight reward model for scaling agent performance. Rather than running expensive test-based evaluation for every candidate solution, the classifier can provide fast approximate scoring to filter or rank solutions. This enables best-of-N sampling strategies where multiple solution attempts are generated and the highest-scoring candidate (according to the reward model) is selected for final evaluation. Future work should investigate using the classifier scores as rewards for reinforcement learning fine-tuning of the underlying LLM.

Heuristic-Based Metrics and Learned Alternatives. Several metrics in the current framework rely on hand-crafted heuristics that could be improved with learned models:

- **Reasoning Quality:** Currently detected via keyword matching (e.g., “I think”, “because”, “let me verify”). A learned classifier trained on human-annotated reasoning quality could better distinguish genuine analytical thinking from superficial pattern matching.
- **Exploration Efficiency:** Computed by comparing explored files to “relevant files” hints, which may not capture all valid exploration paths. A learned model could score exploration quality based on whether the agent gathered sufficient context for the fix.
- **Trajectory Efficiency:** The “optimal trajectory” is estimated heuristically as $2 \times |\text{files}| + 1$ (read, write, submit). This ignores that some fixes require iterative refinement or that reading related files aids understanding. A learned trajectory scorer could better assess whether steps contributed to the solution.
- **Semantic Correctness:** Currently uses AST node overlap, function/class name matching, and proximity heuristics. A code-aware neural model (e.g., CodeBERT, StarCoder embeddings) could provide more robust semantic similarity that captures functional equivalence beyond surface patterns.
- **Failure Mode Classification:** Rule-based thresholds (e.g., “stuck if ≥ 3 repeated actions”) are brittle. A learned classifier could identify failure modes from trajectory patterns with higher accuracy.
- **Patch Similarity:** Line-level diff comparison misses semantically equivalent changes (reordered statements, renamed variables). Neural code similarity models could better assess whether two patches implement the same fix.

A unified learned reward model trained on (trajectory, outcome) pairs could potentially replace many of these heuristics, providing end-to-end scoring that captures complex interactions between metrics that hand-crafted rules miss.

Sampling and Search Strategies. The current framework evaluates single-shot agent runs. More sophisticated sampling strategies could improve resolve rates:

- **Best-of-N sampling:** Generate N independent solution attempts and select the best according to classifier score or heuristic metrics
- **Beam search:** Maintain multiple partial trajectories and prune low-scoring branches
- **Tree search with backtracking:** Allow the agent to backtrack from dead ends and explore alternative paths
- **Temperature scheduling:** Vary sampling temperature across exploration vs. implementation phases

Prompt Optimization. The current system prompt and tool descriptions are hand-crafted based on intuition and limited iteration. Systematic prompt optimization could significantly improve agent performance:

- **Automated prompt search:** Use techniques like DSPy, OPRO, or APE to automatically discover more effective system prompts by optimizing against the success classifier or resolve rate
- **Task-adaptive prompts:** Learn to select or generate prompts conditioned on task characteristics (e.g., bug type, repository structure, difficulty level)
- **Few-shot example selection:** Automatically select the most relevant few-shot examples from a library of successful trajectories based on task similarity
- **Tool description refinement:** Optimize tool descriptions to reduce misuse patterns (e.g., using `write_file` instead of `str_replace_in_file`)
- **Chain-of-thought elicitation:** Experiment with different reasoning scaffolds to improve agent planning and error diagnosis
- **Prompt ensembling:** Combine outputs from multiple prompt variants using the success classifier for selection

The behavioral metrics collected by this framework provide a rich signal for prompt optimization—not just whether a prompt leads to task resolution, but *how* it affects exploration patterns, reasoning quality, and failure modes. This enables more targeted prompt improvements than optimizing for binary success alone.

Parallelization. The current benchmark runner executes tasks sequentially, which limits throughput when evaluating across many tasks and models. Key opportunities for parallelization include:

- **Task-level parallelism:** Run multiple tasks concurrently with isolated repository clones
- **Model-level parallelism:** Evaluate different models on the same task simultaneously
- **Distributed execution:** Distribute benchmark runs across multiple machines for large-scale evaluation
- **Async API calls:** Pipeline LLM API calls to reduce idle time during tool execution

Additional Future Directions.

- Expand task collection to additional languages (JavaScript, Rust, Go) and test frameworks
- Develop real-time interventions that detect failure patterns mid-execution and adjust agent behavior
- Integrate with continuous integration systems for automated regression detection
- Explore multi-agent collaboration where specialized agents handle exploration, implementation, and verification

5.7 Application: Best-of-N Selection

A practical application of the success classifier is **Best-of-N selection**: given N independent agent attempts on the same task, select the attempt most likely to succeed. We evaluated this strategy via simulation using the classifier’s probability estimates.

Methodology. For each $N \in \{1, \dots, 10\}$, we simulated 10,000 trials by sampling N items from our dataset (with replacement) and applying selection strategies: *Random* (uniform selection), *Probability* (select highest $P(\text{success})$), and *Oracle* (select any successful attempt if one exists, representing the upper bound).

Table 21: Best-of-N selection: success rate by strategy

N	Random	Probability	Oracle	Δ vs Random
1	18.4%	18.4%	18.4%	+0.0%
3	18.8%	43.3%	45.4%	+24.6%
5	18.7%	59.2%	64.1%	+40.5%
10	18.1%	77.9%	86.6%	+59.8%

Key findings:

- With $N = 3$, probability selection more than doubles success rate (43.3% vs 18.8%)
- With $N = 5$, success rate triples to 59.2%, compared to $\sim 19\%$ for random selection
- The classifier achieves 92% of oracle performance, indicating good but imperfect calibration

Robustness to classifier noise. We tested degraded classifiers by adding Gaussian noise to probability estimates ($P_{\text{noisy}} = P + \mathcal{N}(0, \sigma)$, clipped to $[0, 1]$). Even with $\sigma = 0.4$, probability selection achieves 38.2% success (vs 18.6% random), demonstrating the approach remains effective with imperfect classifiers.

Compute efficiency. To reach 80% success rate, probability selection requires approximately $N = 10$ attempts, while random selection cannot reach this threshold. The oracle achieves 80% at $N = 8$.

5.8 Application: Behavioral Fingerprinting

Beyond aggregate metrics, we can characterize each model’s “behavioral personality” by examining patterns across multiple dimensions. We compute mean feature vectors per model and aggregate into six behavioral dimensions: *reasoning*, *exploration*, *implementation*, *verification*, *efficiency*, and *robustness*.

Table 22: Behavioral dimension scores by model (0–1 scale, higher = better)

Model	Reas.	Expl.	Impl.	Verif.	Effic.	Robust.
Claude Opus 4	0.11	0.00	0.25	0.12	0.75	0.75
GPT-4o	1.00	0.74	0.85	0.71	0.81	0.60
GPT-5.1	0.00	0.74	0.69	0.60	0.49	0.40
o4-mini	0.00	0.86	0.33	0.48	0.37	0.62

Key findings:

- **Distinct behavioral profiles:** GPT-4o dominates in reasoning (1.00) and implementation (0.85), explaining its 26.9% success rate. Claude Opus 4, despite 0% success, shows

high efficiency (0.75)—it fails fast rather than exploring thoroughly. *Note: Claude Opus 4 data appears incomplete; 0% success with 100% “no changes” suggests configuration issues or interrupted runs.*

- **Exploration patterns:** o4-mini leads exploration (0.86) but has low implementation (0.33), suggesting it explores well but struggles to convert findings into solutions.
- **Model similarity:** GPT-4o and GPT-5.1 are most similar (cosine: 0.01), while Claude Opus 4 and GPT-5.1 are most different (-0.63).

This fingerprinting enables targeted model selection: choose GPT-4o for tasks requiring strong reasoning and implementation, o4-mini for exploration-heavy tasks, or use ensemble approaches combining complementary strengths.

5.9 Application: Reward Modeling

The success classifier naturally provides a **reward signal** for reinforcement learning. Instead of sparse binary feedback (pass/fail at episode end), the classifier outputs $P(\text{success}) \in [0, 1]$ as a dense, continuous reward.

Reward signal quality. We evaluate the classifier’s utility as a reward model by measuring separation between success and failure distributions:

Table 23: Reward signal quality metrics

Metric	Successes	Failures
Mean reward	0.539	0.103
Std deviation	0.150	0.135
Separation	0.435	
Distribution overlap	0.097	

The separation of 0.435 indicates the reward model provides meaningful discriminative signal on real data. The low overlap (9.7%) means successful and failed trajectories receive substantially different rewards, enabling effective ranking.

Reward formulation comparison. We compare four reward formulations:

- *Sparse:* Binary 0/1 based on final outcome (variance: 0.149)
- *Dense:* Continuous $P(\text{success})$ from classifier (variance: 0.047)
- *Shaped:* $P(\text{success}) + 0.1 \cdot \mathbf{1}[P > 0.8]$ (variance: 0.047)
- *Advantage:* $P(\text{success}) - \bar{P}$ (variance: 0.047)

Dense rewards exhibit $3\times$ lower variance than sparse rewards, providing more stable learning signal. Additionally, dense rewards provide signal throughout the trajectory rather than only at termination, enabling more effective credit assignment.

API. The reward model is packaged as a Python class:

```
rm = RewardModel.load('reward_model.pkl')
reward = rm.score(trajecory_features) # -> [0, 1]
```


5.10 Application: Curriculum Learning Signal

Not all tasks are equally valuable for training. We analyze task **informativeness**—how much a task contributes to learning the decision boundary between success and failure.

Methodology. For each task, we compute:

- *Success rate*: Fraction of models that solved it
- *Model agreement*: 1 if unanimous, 0 if perfectly split
- *Classifier uncertainty*: $1 - |2P - 1|$ (highest at $P = 0.5$)
- *Informativeness*: Uncertainty \times (1 - agreement)

Table 24: Task difficulty distribution (52 tasks, 4 models each)

Category	Count	Success Rate	Model Agreement
Easy ($\geq 75\%$)	5	75–100%	High (unanimous)
Medium (25–75%)	10	25–75%	Mixed
Hard ($\leq 25\%$)	37	0–25%	High (all fail)

Key findings:

- **Most tasks are hard**: 71% of tasks (37/52) have $\leq 25\%$ success rate
- **Model disagreement is informative**: 18 tasks show disagreement between models—these are prime candidates for curriculum learning
- **Boundary tasks**: Tasks with 50% success rate (e.g., scikit-learn-30395, scikit-learn-30103) are most informative for training

Curriculum ordering. For training, we recommend: (1) Easy tasks first to build confidence, (2) Medium/boundary tasks to learn decision boundaries, (3) Hard tasks last for edge cases. Within each category, prioritize by informativeness.

5.11 Application: Contrastive Analysis

We identify which behavioral features most distinguish successful from failed trajectories using effect size analysis (Cohen’s d).

Table 25: Top discriminative features (Cohen’s d , significant at $p < 0.05$)

Feature	Cohen’s d	Effect	Direction
tool_errors_occurred	−1.64	large	↓ in success
no_changes_made	−1.33	large	↓ in success
used_str_replace	+1.17	large	↑ in success
submitted	+1.11	large	↑ in success
trajectory_efficiency	+0.95	large	↑ in success
correct_files_touched	+0.94	large	↑ in success

Recipe for success:

- **DO**: Use str_replace for edits, submit solutions, maintain high trajectory efficiency, touch correct files, read before writing

- **AVOID:** Tool errors (strongest predictor of failure), making no changes, modifying wrong files

The contrastive analysis reveals that *how* agents work matters as much as *what* they produce. Agents that avoid tool errors and actually make changes succeed; those that get stuck in exploration loops fail.

5.12 Application: Targeted Task Generation

We use the weakness analysis to automatically generate task specifications that target specific model deficiencies.

Model weakness profiles. Analysis reveals distinct failure patterns:

- **Claude Opus 4:** 100% of failures involve making no changes; 98% show low exploration efficiency
- **GPT-4o:** 95% of failures involve tool errors; 53% skip verification
- **GPT-5.1:** 97% of failures involve tool errors
- **o4-mini:** 100% of failures involve tool errors; 63% make no changes

Generated assets. We create 8 task templates across categories (exploration, implementation, debugging, verification) and 15 targeted task specifications. Each task includes difficulty rating, required skills, estimated steps, and rationale linking to the specific weakness being tested.

Sample template: “Deep Codebase Navigation” (hard) targets low exploration efficiency by requiring agents to trace dependencies across multiple files to find a bug whose symptoms appear far from the root cause.

5.13 Application: Cross-Repository Transfer

We analyze how well behavioral features generalize across different task groups, simulating cross-repository transfer.

Methodology. We split the 52 tasks into 4 pseudo-repositories and measure: (1) feature stability across groups, (2) train-on-one/test-on-another performance, and (3) leave-one-task-out cross-validation.

Key findings:

- **40/59 features transfer well:** These “universal” features show low variance across task groups (stability score < 0.5)
- **Transfer gap is small:** Same-group accuracy (99%) vs cross-group accuracy (84%) shows only 15% degradation
- **Strong generalization:** Leave-one-task-out CV achieves $94.2\% \pm 13.5\%$ accuracy

Most transferable features: `tool_errors_occurred`, `submitted`, `used_str_replace`, `ran_tests`—these behavioral signals generalize across task types.

Repo-specific features: `exploration_steps`, `files_explored`—these depend on repository structure and may need recalibration.

5.14 Interactive Dashboard

We provide an interactive HTML dashboard that visualizes all analysis results in a single interface. The dashboard includes:

- Model performance comparison with success rates and behavioral fingerprints
- Best-of-N selection curves showing probability vs random vs oracle strategies
- Contrastive analysis with effect sizes and “recipe for success”
- Curriculum learning task difficulty distribution
- Cross-repository transfer performance matrix
- Reward model signal quality visualization

The dashboard requires no server—it runs entirely in-browser using Plotly.js for interactive charts. This enables easy sharing of evaluation results with stakeholders.

6 Conclusion

We presented a comprehensive framework for evaluating coding agents that goes beyond binary success/failure metrics. Our evaluation of four models on 52 scikit-learn tasks revealed several key findings:

- **Performance varies significantly:** Resolve rates ranged from 34.6% (GPT-5.1) to 11.5% (o4-mini), demonstrating substantial room for improvement in current coding agents.
- **Efficiency predicts success:** Successful agents used fewer steps (8.2 vs 12.4), completed faster (22s vs 90s), and made fewer tool errors (1.2 vs 5.8). Extended deliberation did not improve outcomes.
- **Early localization is critical:** Agents that found correct files within 5 steps succeeded 68% of the time vs 12% for those needing more than 10 steps.
- **Submission confidence matters:** GPT-5.1’s 98% submission rate correlated with higher resolve rates compared to o4-mini’s cautious 35%.
- **Failure modes differ by model:** GPT-5.1 primarily misunderstood issues, GPT-4o gave up early, and o4-mini explored excessively.

The framework provides:

- Task collection from GitHub PRs
- Multi-provider agent implementation
- Comprehensive behavioral metrics across 9 categories
- Success prediction classifier
- Visualization and reporting tools

The success classifier opens opportunities for using behavioral metrics as reward signals for scaling agent performance through best-of-N sampling and reinforcement learning. Combined with parallelized evaluation infrastructure, this framework provides a foundation for systematic improvement of coding agents on real-world software engineering tasks.

Code Availability

The complete framework is available at: [https://github.com/\[REPOSITORY\]](https://github.com/[REPOSITORY])

A Additional Metrics

These metrics are computed when running with the `-detailed-metrics` flag.

A.1 Phase Distribution Metrics

Phase Classification: Each tool call is categorized:

$$\text{Phase}(a) = \begin{cases} \text{EXPLORATION} & \text{if } a \in \{\text{read}, \text{search}, \text{list}\} \\ \text{IMPLEMENTATION} & \text{if } a \in \{\text{write}, \text{str_replace}\} \\ \text{VERIFICATION} & \text{if } a = \text{run_tests} \end{cases} \quad (16)$$

Phase Percentages: $p_\phi = |\{a_t : \text{Phase}(a_t) = \phi\}|/T$

Read-Before-Write (RBW): $\mathbb{K}[\exists t_r < t_w : a_{t_r} = \text{read}(f) \wedge a_{t_w} = \text{write}(f)]$

Test-After-Change (TAC): $\mathbb{K}[\exists t_w < t_t : a_{t_w} \in \{\text{write}\} \wedge a_{t_t} = \text{run_tests}]$

A.2 Convergence Metrics

Progress Curve: $\mathbf{S} = (S_1, S_2, \dots, S_T)$ where $S_t = \text{DiffSimilarity}(P_t, P_{\text{gold}})$

Monotonic Progress: $\mathbb{K}[\forall t > 1 : S_t \geq S_{t-1}]$

Progress Volatility: $\sigma_S = \sqrt{\frac{1}{T-1} \sum_{t=2}^T (S_t - S_{t-1})^2}$

Had Regression: $\mathbb{K}[\exists t : S_t < S_{t-1}]$

A.3 Error Recovery Metrics

Total Errors: $N_{\text{errors}} = |\{a_t : \text{result}(a_t) = \text{error}\}|$

Recovery Rate: $r_{\text{recovery}} = |\{e : \text{recovered}(e)\}|/(N_{\text{errors}} + \epsilon)$

Stuck Episodes: Sequences where $a_t \approx a_{t+1} \approx \dots \approx a_{t+k}$

Maximum Stuck Duration: $D_{\text{stuck}} = \max_k |\text{Stuck}_k|$

A.4 Tool Usage Metrics

Total Tool Calls: $N_{\text{tools}} = |\{a_t : a_t \text{ is a tool call}\}|$

Tool Distribution: Boolean flags for `read_relevant_files`, `used_str_replace`, `used_write_file`, `ran_tests`, `submitted`

Tool Error Count: $N_{\text{tool_errors}} = |\{a_t : \text{result}(a_t) \text{ indicates error}\}|$

A.5 Patch Quality Metrics

File Precision/Recall:

$$P_{\text{files}} = \frac{|\text{Files}_{\text{agent}} \cap \text{Files}_{\text{gold}}|}{|\text{Files}_{\text{agent}}|}, \quad R_{\text{files}} = \frac{|\text{Files}_{\text{agent}} \cap \text{Files}_{\text{gold}}|}{|\text{Files}_{\text{gold}}|} \quad (17)$$

Lines Added/Removed: Count of added and deleted lines in agent patch

Patch Size Ratio: $\rho_{\text{size}} = |P_{\text{agent}}|/|P_{\text{gold}}|$

A.6 Semantic Correctness Metrics

Location Score: Overlap between modified code regions

$$L_{\text{score}} = \frac{\sum_{f \in \text{Files}_{\text{gold}}} \sum_{r \in \text{Regions}_f} \mathbb{I}[\text{agent modifies near } r]}{|\text{Regions}_{\text{gold}}|} \quad (18)$$

Same Function Modified: $\mathbb{I}[\text{Funcs}_{\text{agent}} \cap \text{Funcs}_{\text{gold}} \neq \emptyset]$

Change Type Match: Whether agent and gold patches have same change type (add/remove/modify)

Likely Correct: $\text{TestsPass} \vee (S_{\text{semantic}} \geq 0.6 \wedge \text{SameFile} \wedge \text{SameFunc})$

A.7 Failure Mode Taxonomy

Table 26: Failure mode detection criteria

Category	Mode	Criterion
Exploration	Excessive Exploration	$p_{\text{EXP}} > 0.7$
	Insufficient Context	$ \text{Files}_{\text{explored}} < 2$
Understanding	Misunderstood Issue	Keywords overlap < 0.2
Implementation	Wrong Location	Wrong file modified
	Incomplete Fix	$0.3 < S < 0.7$
Process	Stuck in Loop	$D_{\text{stuck}} \geq 3$
	No Submission	<code>submit</code> not called

B Complete Feature List

Table 27: All features used in success prediction classifier

Category	Features
Reasoning (7)	reasoning_quality_score, has_explicit_reasoning, mentions_issue_keywords, mentions_relevant_files, hypothesizes_before_acting, explains_changes, verifies_after_change
Phases (9)	exploration_steps, implementation_steps, verification_steps, exploration_pct, implementation_pct, verification_pct, phase_transitions, followed_read_before_write, followed_test_after_change
Exploration (6)	files_explored, directories_explored, relevant_file_discovery_step, exploration_efficiency, wasted_explorations, search_to_read_ratio
Trajectory (4)	trajectory_length, optimal_length, trajectory_efficiency, unnecessary_steps
Convergence (6)	final_similarity, max_progress, converged, monotonic_progress, had_regression, progress_volatility
Error Recovery (6)	total_errors, recovered_errors, recovery_rate, max_repetition, stuck_episodes, max_stuck_duration
Tool Usage (7)	total_tool_calls, read_relevant_files, used_str_replace, used_write_file, ran_tests, submitted, tool_errors_count
Patch Quality (6)	correct_files_touched, lines_added, lines_removed, patch_too_large, steps_per_file, edit_to_explore_ratio
Semantic (8)	fixes_same_file, fixes_same_function, fixes_same_class, fixes_same_code_region, location_score, change_type_match, modifies_same_variable, modifies_same_call