CODETASTE: Can LLMs Generate Human-Level Code Refactorings?

Alex Thillen [1] **Niels Mündler** [1] **Veselin Raychev** [2] **Martin Vechev** [1]

Abstract

Large language model (LLM) coding agents can generate working code, but their solutions often accumulate complexity, duplication, and architectural debt. Human developers address such issues through refactoring: behavior-preserving program transformations that improve structure and maintainability. In this paper, we investigate if LLM agents (i) can execute refactorings reliably and (ii) identify the refactorings that human developers actually chose in real codebases. We present CODETASTE, a benchmark of refactoring tasks mined from large-scale multi-file changes in open-source repositories. To score solutions, we combine repository test suites with custom static checks that verify removal of undesired patterns and introduction of desired patterns using dataflow reasoning.

Our experimental results indicate a clear gap across frontier models: agents perform well when refactorings are specified in detail, but often fail to discover the human refactoring choices when only presented with a focus area for improvement. A propose-then-implement decomposition improves alignment, and selecting the best-aligned proposal before implementation can yield further gains. CODETASTE provides an evaluation target and a potential preference signal for aligning coding agents with human refactoring decisions in realistic codebases.

1. Introduction

Coding agents are being rapidly adopted across the software engineering industry (Mürtz & Müller, 2025; Sarkar, 2025), mostly fueled by their recent impressive performance on autonomous resolution of real-world software engineering tasks (Jimenez et al., 2024; Badertdinov et al., 2025). However, success on issue resolution does not imply that agents can sustain code quality across iterations. SlopCodeBench (Orlanski et al., 2024) evaluates iterative specification updates and measures verbosity and structural erosion, showing that coding-agent-generated code becomes longer and more structurally strained as requirements evolve. A recent effort to build a browser rendering engine using autonomous coding agents (Wilson Lin, 2026) illustrates some practical consequences: despite making notable early progress, the agents effectively stalled, unable to continue extending a codebase they had themselves created. Similarly, an experiment tasking a team of agents to autonomously build a C compiler (Carlini, 2026) produced a functional 100,000-line codebase capable of compiling the Linux kernel, but resulted in rigid, inefficient code that lacked human-like abstraction. These observations suggest that while current methods excel at producing functional patches, they struggle to produce code that remains comprehensible and extensible over time.

Lack of challenging refactoring benchmarks Refactoring is the software engineering practice of applying behavior-preserving transformations that improve structure and maintainability. OpenAI reports that they train and evaluate their models on large-scale code refactoring tasks, without disclosing their internal evaluation frameworks (OpenAI, 2025a). The academic community has also contributed benchmarks for refactoring: Refactor-Bench provides few-file tasks with syntactic tests (Gautam et al., 2025), and SWE-Refactor collects 1,099 refactoring of a fixed set of method-level refactorings from 18 Java repositories (Xu et al., 2026).

These benchmarks have a key limitation: they only cover refactorings of small and limited scope, which results in them quickly becoming obsolete. For example, GPT-5 CODEX already achieves 75% accuracy on SWE-Refactor. Beyond this size limitation, these benchmarks do not measure whether the model can determine autonomously which refactoring should be performed. This distinction matters: to bootstrap a code base out of accumulated technical debt, agents must recognize when code has become unmaintainable and determine what transformations are necessary to restore clarity (Perry & Wolf, 1992). A recent related study

---

[1] Department of Computer Science, ETH Zurich, Zurich, Switzerland [2] LogicStar.ai, Zurich, Switzerland. Correspondence to: Alex Thillen <athillen@ethz.ch>, Niels Mündler <niels.muendler@inf.ethz.ch>, Veselin Raychev <veselin@logicstar.ai>, Martin Vechev <martin.vechev@inf.ethz.ch>.

Preprint. March 4, 2026.

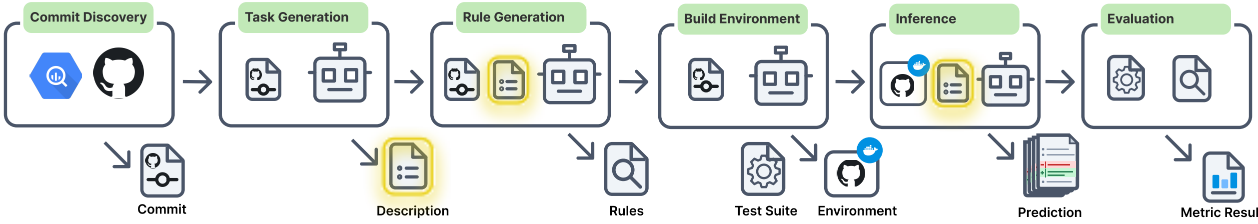


Figure 1. Overview of the CODETASTE benchmark pipeline. The workflow follows six key stages: (1) Commit Discovery from GitHub; (2) Generation of Task Descriptions; (3) Rule Generation of static analysis rules that capture refactoring intent; (4) Build Environment to run test suite and agent inference in containerized environment; (5) Inference where the agent generates a patch; and (6) Evaluation of generated patches against static analysis rules and test suite.

found that LLMs rarely identify refactoring opportunities on their own: given Java files as input, ChatGPT identified only 28 out of 180 refactoring opportunities that human experts had selected (Liu et al., 2025).

Our work In this work, we investigate this challenge extensively and use precise code-level measurements to assess whether state-of-the-art coding agents can autonomously discover and implement relevant refactorings in real-world code bases. We introduce an alignment score that measures how many occurrences of undesired coding patterns are converted to desired versions, counting only functionally correct implementations. Our results indicate a clear gap: while frontier models score highly on implementing complex refactorings that are specified in detail, achieving up to 70% alignment in our evaluation, they fail to mitigate undesired coding patterns correctly when no concrete refactoring is specified, achieving overall less than 8% alignment. We introduce CODETASTE, a benchmark designed to measure this gap.

A scalable pipeline for discovering refactoring instances. We mine popular GitHub repositories for 100 large multi-file refactorings performed by human developers. For each task, we generate an executable environment and static analysis rules that match the changed code patterns. The latter support semantic dataflow reasoning within files, which provides more flexibility than syntactic matching, and is automatically verifiable. After the model applies a refactoring, we measure performance, ensuring that (i) the repository test suite passes and (ii) the static analysis rules confirm code pattern transformation.

Two complementary tracks for evaluating agent performance. Our benchmark features two tracks: In the *Instructed track* the agent is tasked to correctly implement a detailed refactoring. Frontier models achieve alignment scores of up to 69.6%, indicating both correct implementations and consequent transformation. The *Open track* evaluates whether agents can propose refactorings that align with human developer choices. In this track, the agent is only tasked to improve a general focus area. We discover that agents struggle significantly here, achieving only single-digit scores, even with oracle support.

We observe that frontier models consistently outperform lower capability models on our benchmark, with for example GPT-5 CODEX achieving 7.7% alignment in the open track direct setting compared to 2.3% for QWEN3. This suggests that human refactoring preferences are already partially captured by general model training. Still, our case studies, we observe that current agents often take shortcuts and tend towards apply minimal, insufficient changes, unless explicitly instructed to plan out a refactoring.

2. Methodology

In Figure 1, we provide a summary of the CODETASTE pipeline. Based on a dataset of historic code changes, we (i) mine large, multi-file refactorings performed by human developers, (ii) translate each refactoring into task descriptions for our two tracks, (iii) build a reproducible execution environment that can run each repository's unit tests, and (iv) generate static analysis rules that capture desired and undesired patterns of the transformation. We then run agents to produce a candidate patch and evaluate it using both the repository test suite and our static rules.

During evaluation, we task coding agents to reimplement the original human refactoring. These agents can interact with the task description, the codebase, and the test suite, and the execution environment, producing a final predicted patch. Finally, we evaluate this patch using (i) the test suite to validate whether the functionality has been preserved and (ii) the static analysis rules to validate how consistently the refactoring was implemented.

2.1. Notation and Definitions

We first introduce the notation to describe codebases R, their test suites T, and changes to these codebases in the

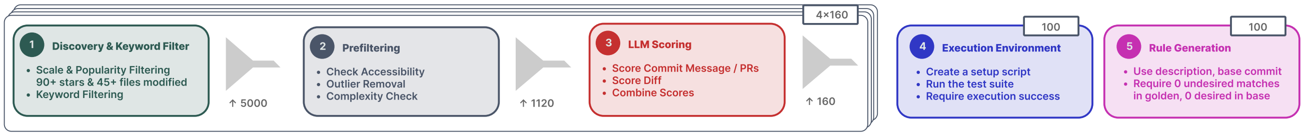


Figure 2. Data funnel for obtaining the CODETASTE benchmark instances. In each step of the main commit pipeline, we filter for the top n entries using different scoring schemes. Finally, we extract 100 instances for which our agentic setup produced a working execution environment.

form of patches X. Following the notation of Mündler et al. (2024), we denote as $R \circ X$ a codebase R after applying patch X. In this case, the human-authored *base commit* is R. The developer written *golden patch* is denoted X^*, and the corresponding *golden state* or *golden commit* is $R \circ X^*$.

On any codebase R, a set of tests T can be used to check the correctness of the applied patch. Executing a test suite T of repository R runs a total number of tests $|T|$ and returns $\text{exec}_R(T) = (P_R, F_R)$ a count of passing tests P_R and failing tests F_R. If execution of the test suite fails unexpectedly, e.g., due to a crash, we set $(P_R, F_R) = (0, 0)$.

Let L_R be the set of lines in R, where each $l \in L_R$ is a triple (f, n, s) representing its file, line number in the file, and line content, in this order. A patch X induces a partial mapping $\mathcal{M}_X : L_R \rightharpoonup L_{R \circ X}$ tracking line persistence. Two lines $l \in L_R$ and $l' \in L_{R \circ X}$ are *unchanged* in patch X ($l \equiv_X l'$) iff $\mathcal{M}_X(l) = l'$. Lines deleted by X are excluded from $\text{dom}(\mathcal{M}_X)$, while new lines in $R \circ X$ lack a preimage. Edited lines are represented by a deleted line and a new line, respectively. We define the sets of *added* lines $L_X^+ = L_{R \circ X} \setminus \text{im}(\mathcal{M}_X)$, the *removed* lines $L_X^- = L_R \setminus \text{dom}(\mathcal{M}_X)$ and *changed* lines $L_X = L_X^+ \cup L_X^-$.

3. Benchmark Construction

In this section we outline our benchmark construction process, sketched in Figure 2.

3.1. Data curation

Each task instance in our benchmark corresponds to a commit in a codebase. We obtain a list of commits for further processing using a three-stage pipeline. We first query and keyword-filter candidate repositories and commits to prioritize 5000 commits that perform large refactorings. Second, we prefilter to around 1000 commits by removing inaccessible repositories, documentation-only changes, and outlier diffs. Third, we score commit messages and diffs with a small LLM and keep the highest-ranked 160 candidates. We run these three steps four times, once for each of the years 2023, 2024, and 2025 from the GitHub Archive (Grigorik, 2011), and once for the GitHub Activity Data dataset (GitHub, 2016). The full details on the pipeline can be found in Appendix A.1.

Our current pipelines and filters are designed to generate a benchmark of relatively small size, due in large part to the high cost of performing a refactoring, often exceeding 10 USD per task. Our collection using can be scaled up trivially by leveraging additional base databases and modifying its filter criteria.

3.2. Task generation

For each task instance, we generate a structured, highly-detailed task description modeled after a human-written GitHub issue using an LLM. The description is synthesized from the commit message, associated Pull Requests, including titles, bodies, and linked issues, and the code of the golden patch X^*.

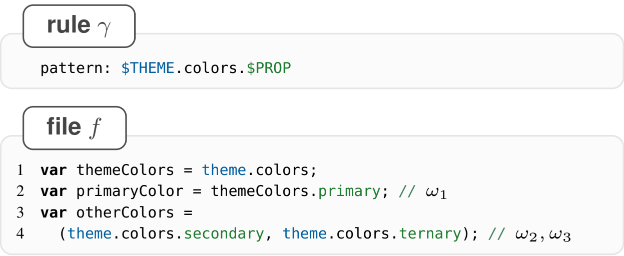
The LLM is instructed to structure the task description as three components: 1) *Title:* A brief heading for the refactoring task. 2) *Summary:* A concise overview of the technical changes, and 3) *Why:* The rationale and relevance of the refactoring, with the goal of providing a highly detailed task description structured similarly to the instructions a human could write for a coding agent. The prompt used for this generation can be found in Figure 13.

3.3. Build Environment

To provide a stable and reproducible execution environment, our method utilizes a multi-phase bootstrap process that pairs each benchmark instance with a specialized containerized environment.

Base Container The foundation is an Ubuntu-based container image with pre-configured toolchains across language ecosystems (details in Appendix A.2).

Phase 1: Agentic Environment Setup For each instance, our method spins up a base container and performs a shallow clone of R and $R \circ X^*$. A coding agent is initialized inside the container with administrative privileges, time and expense limit. The agent is tasked with installing any additionally required system dependencies and creating scripts that allow reproducing a standardized execution environment for the repository. We provide the prompt in the Appendix in Figure 11 and an example excerpt of the generated scripts in Figure 12.



```
pattern: $THEME.colors.$PROP
```

```
1  var themeColors = theme.colors;
2  var primaryColor = themeColors.primary; // ω1
3  var otherColors =
4    (theme.colors.secondary, theme.colors.ternary); // ω2, ω3
```

Figure 3. Example OpenGrep rule $\gamma \in \Gamma$ with symbolic propagation. In repository $R = \{f\}$, the two matching witnesses $\Omega(\gamma, R) = \{\omega_1, \omega_2\}$, include aliased and direct `.colors` property access. Even though match count $M(\gamma, R) = 3$, coverage $\mathcal{C}(\gamma, R) = 2$, as Line 4 contains several matches.

Validation and Quality Gates Upon completion of the environment setup phase, the framework validates the reproducibility of the environment by executing $\text{exec}_R(T)$ and $\text{exec}_{R \circ X^*}(T)$ using the standardized execution scripts. A bootstrapping process is deemed successful only if the following conditions are met:

$$\forall R' \in \{R, R \circ X^*\}. \quad |T_{R'}| \geq 10 \quad \text{and} \quad \frac{P_{R'}}{|T_{R'}|} \geq \frac{3}{10}$$

Phase 2: Runtime Creation After validation, the resulting image is transformed into a hardened runtime to prevent leaking golden solutions and standardize the interface for model evaluation. First, by removing history and network accesses, we ensure that the golden state $R \circ X^*$ is inaccessible from the base state R during inference. Second, we add a unified entrypoint script which serves as the primary orchestration layer. This script is executed upon container start and enables execution of agents under restricted network access, extracting the generated patch $\hat{X}$ for later evaluation.

3.4. Rule Generation

To capture the semantic intent of a code transformation without overfitting to specific concrete changes, we frame rule generation as a search task performed by an LLM-based agent. The agent is only provided the task description generated in the previous step, to avoid overfitting to the actual code changes in X^*. For each task instance, the agent is then tasked to generate a set of static analysis rules $\Gamma = \Gamma^+ \cup \Gamma^-$ using OpenGrep syntax. These rules use patterns over the AST with optional intra-file dataflow (e.g., symbolic propagation and taint mode) to describe the transformation abstractly.

Rule Formalization A rule γ produces a multiset of witnesses on a codebase R, denoted $\Omega(\gamma, R) = \{\omega_1, \ldots, \omega_k\}$, where each witness $\omega \subseteq L_R$ is a subset of lines from R that together satisfy the pattern specified in rule γ. From these, we derive two metrics: (i) match count $M(\gamma, R) = |\Omega(\gamma, R)|$, representing the number of times a given rule γ matches in codebase R, and (ii) line coverage $\mathcal{C}(\gamma, R) = \bigcup_{\omega \in \Omega(\gamma, R)} \omega \subseteq L_R$, representing the subset of all lines in R that are covered by γ. We present an abbreviated example with match count and coverage in Figure 3, and the unabbreviated example in Appendix B.

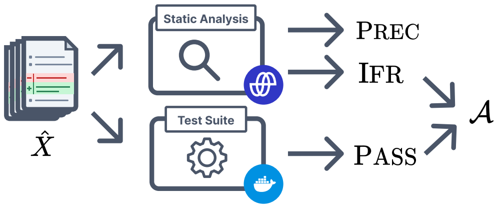


Figure 4. Sketch of the evaluation metrics for CODETASTE.

We categorize rules into additive (Γ^+) patterns introduced by the refactor (X^*) and absent in the base codebase (R):

$$\forall \gamma \in \Gamma^+ : M(\gamma, R \circ X^*) > 0 \wedge M(\gamma, R) = 0$$

and reductive (Γ^-) patterns removed from R that are absent in $R \circ X^*$:

$$\forall \gamma \in \Gamma^- : M(\gamma, R) > 0 \wedge M(\gamma, R \circ X^*) = 0$$

Agentic Discovery Loop The agent runs in two distinct passes to discover Γ^+ and Γ^- independently. It follows an iterative loop using three tools: (i) allowing read-only shell commands in the base codebase R, (ii) registering new additive / reductive rules with an automated feedback loop on whether the addition is valid and distinct, and (iii) a submission command to stop the interaction. We detail the available tools in Appendix A.3.

3.5. CODETASTE

We apply the described pipeline, using SONNET 4.5 to synthesize task descriptions, set up build environments and generate static rules. We thus obtain CODETASTE, our main benchmark, consisting of 100 instances distributed across 87 repositories and 6 programming languages. The detailed distribution across languages is shown in Figure 5. The benchmark comprises complex and large refactorings, as shown in Table 1, with instances on average requiring edits to 91.52 files and 2605.39 lines of code. The most complex tasks require up to 18821 line changes across up to 290 files. To validate correctness, the benchmark runs an average of 1638.53 tests and validates an average of 29.66 additive and 63.41 reductive rules per instance.

During inference, the benchmark evaluates coding agents on the base commit R within the pre-configured execution environment, where they are provided with a refactoring task. The agents are free to navigate and modify the entire

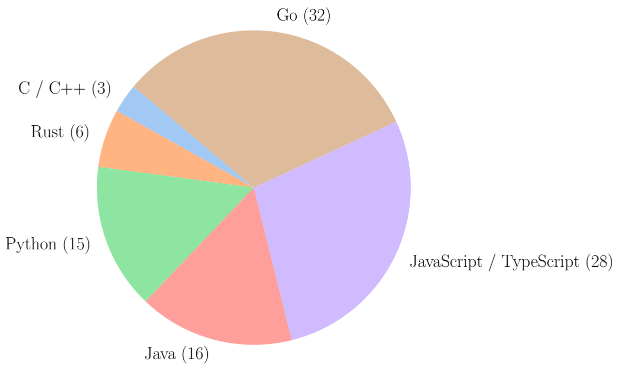


Figure 5. Programming language distribution in the selected 100 instances of CODETASTE

environment, but have limited access to the internet to prevent them from obtaining solutions from the original repository. Upon completion, we extract a generated patch $\hat{X}$ from the modifications to the repository state.

Tracks Regarding the refactoring task, the benchmark features two main tracks. In the *Instructed Track*, the model is provided with a detailed description of the refactor to implement. This track mostly evaluates the coding agent's capability to perform long-context tasks consistently. These tasks contain significant detail, as shown in Table 1, with on average 3451 characters per description.

The goal of the *Open Track* is to vaguely define the area of the desired refactoring, or a problematic aspect of the code base, to avoid completely random exploration, but not to reveal directly what is desired as a refactoring. We derive the open track description via a secondary LLM pass over the instructed track full description. In the end, we manually investigated all open track descriptions to confirm that that they indeed do not reveal the actual refactoring. Examples of both description types are shown in Figures 13 and 14. The average description is significantly shorter than in the instructed track, with only 38 characters on average.

Open Track Modes To allow for fine-grained understanding of the refactoring agent decisions in the open track, we define three different modes of inference sketched in Figure 6. In *Direct* mode, the agent implements changes directly from the instruction. In *Plan* mode, the agent first generates a single implementation plan. Then, the agent is reinitialized and prompted to execute this plan in a second pass. In *Oracle Multiplan* mode, the agent generates multiple candidate plans. An LLM-based oracle judge with access to the full refactoring description then selects the plan that is closest to the refactoring. Finally, the agent is reinitialized to execute the selected plan. The used prompts are provided the Appendix in Figures 15–17.

Table 1. Summary statistics for CODETASTE instance metrics.

	MAX.	MEAN
Instruction Length (# Chars)		
Instructed Track	3451.0	1921.02
Open Track	68.0	38.00
Evaluation		
Total Tests	12449	1638.53
Additive Rules	87	29.66
Reductive Rules	120	63.41
Total Rules	170	93.07
Golden Patches		
Files Changed	290	91.52
Lines Added	7172	1190.33
Lines Removed	11649	1415.06
Lines Edited	18821	2605.39

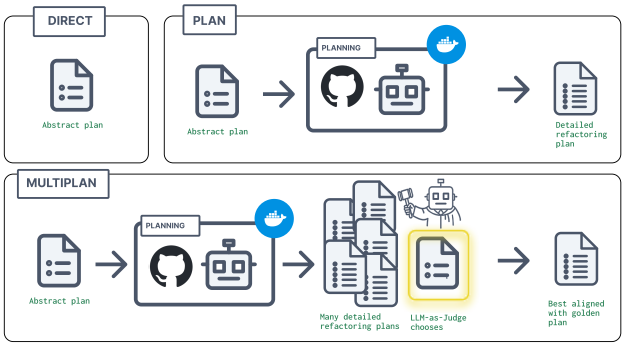


Figure 6. Inference modes used in open track evaluation.

4. Experimental Evaluation

In this section, we report the results of our extensive evaluation of coding agents on CODETASTE.

4.1. Experimental Setup

Models We evaluate the performance of four coding agents across different task specifications. Our model cohort includes GPT-5.2 and GPT 5.1 CODEX MINI (GPT-5.1 M) (Singh et al., 2025), CLAUDE 4.5 SONNET (SONNET 4.5) (Anthropic, 2025), and the instruct version of the open-source QWEN3 CODER 30B A3B (QWEN3) (Yang et al., 2025). Each model is evaluated using its respective model-developer-recommended agent harness: Codex CLI v0.77 (OpenAI, 2025b), Claude Code v2.0.76 (Code Claude Docs, 2025), and Qwen Code v0.6.2 (QwenLM, 2025). For Codex CLI was explicitly configured the reasoning effort to `medium`, while Claude Code and Qwen Code are left to use their respective defaults. We use SONNET 4.5 as the oracle judge for plan selection in Oracle Multiplan mode (Figure 17).

Functional Correctness To evaluate functional correctness of a predicted patch $\hat{X}$, we require no major regression of the test suite. Concretely, we require a minimum number

p_R of test cases to pass on the patched code base $R \circ \hat{X}$, and a maximum number f_R of test cases to fail. To determine p_R and f_R, we execute k times the test suites of both R and $R \circ X^*$. We then set $p_R = \min_{i \in \{1,\dots,k\}}(P_R^i, P_{R\circ X^*}^i)$ and $f_R = \max_{i \in \{1,\dots,k\}}(F_R^i, F_{R\circ X^*}^i)$, where P_R^i denotes the number of passing tests of the i-th run on codebase R, and F_R^i failing tests respectively. Finally, we define the functional correctness score $\textsc{Pass}(\hat{X}) = \mathbb{1}\left[F_{R\circ\hat{X}} \le f_R \wedge P_{R\circ\hat{X}} \ge p_R\right]$. We set $k = 5$ to account for nondeterministic tests (Mündler et al., 2024).

Instruction Following Rate We assess adherence to the refactoring intent using the discovered static analysis rules Γ. Let $\mathbb{1}_M(\gamma, R) = \mathbb{1}[M(\gamma, R) > 0]$ indicate that rule γ matches in state R. The instruction following rate (IFR) measures the recall of additive and reductive rules in the resulting state $R \circ X$: The additive rules (Γ^+) represent desired patterns, i.e., patterns that should match, while the reductive rules (Γ^-) represent undesired patterns that should not match. The additive instruction following scores $\textsc{Ifr}^+$is then the percentage of rules $\gamma \in \Gamma^+$ that match on the predicted state of the repository $R \circ \hat{X}$. The reductive instruction following score $\textsc{Ifr}^-$ is respectively defined as the percentage of non-matching rules. The total instruction following score IFR is a weighted combination of the scores. By construction, $\textsc{Ifr}(X^*) = 1$.

$$\textsc{Ifr}^+(\hat{X}) = \frac{1}{|\Gamma^+|}\sum_{\gamma\in\Gamma^+}\mathbb{1}_M(\gamma, R\circ\hat{X})$$

$$\textsc{Ifr}^-(\hat{X}) = \frac{1}{|\Gamma^-|}\sum_{\gamma\in\Gamma^-}\left(1 - \mathbb{1}_M(\gamma, R\circ\hat{X})\right)$$

$$\textsc{Ifr}(\hat{X}) = \frac{|\Gamma^-|}{|\Gamma|}\textsc{Ifr}^-(\hat{X}) + \frac{|\Gamma^+|}{|\Gamma|}\textsc{Ifr}^+(\hat{X})$$

Alignment Score We combine the instruction following rate and the functional correctness into an *alignment score* $\mathcal{A}$. Concretely, the alignment score $\mathcal{A}(\hat{X}) = \textsc{Pass}(\hat{X}) \times \textsc{Ifr}(\hat{X})$. The multiplication ensures that adherence to refactoring intent is only rewarded if the generated patch is also functionally correct.

Precision Precision $\textsc{Prec}(\hat{X})$ measures the extent to which a patch $\hat{X}$ avoids unrelated changes. As defined in §2.1, a patch $\hat{X}$ defines two sets of lines, $L_{\hat{X}}^-$ and $L_{\hat{X}}^+$ corresponding to additions and removals respectively. For a given codebase R, $\mathcal{C}_\Gamma(R) = \cup_{\gamma\in\Gamma}\mathcal{C}(\gamma, R)$ defines the set of all lines matched by the rule set Γ. We then define additive precision $\textsc{Prec}^+$ as the percentage of added lines $L_{\hat{X}}^+$ that also match the additive rules Γ^+. Negative precision $\textsc{Prec}^-$ is the percentage of removed lines $L_{\hat{X}}^-$ that matched the reductive rules Γ^-. Precision PREC is the weighted average of these terms.

$$\textsc{Prec}^+(\hat{X}) = \frac{|L_{\hat{X}}^+ \cap \mathcal{C}_{\Gamma^+}(R\circ\hat{X})|}{|L_{\hat{X}}^+|}$$

$$\textsc{Prec}^-(\hat{X}) = \frac{|L_{\hat{X}}^- \cap \mathcal{C}_{\Gamma^-}(R\circ\hat{X})|}{|L_{\hat{X}}^-|}$$

$$\textsc{Prec}(\hat{X}) = \frac{|L_{\hat{X}}^+|}{|L_{\hat{X}}|}\textsc{Prec}^+(\hat{X}) + \frac{|L_{\hat{X}}^-|}{|L_{\hat{X}}|}\textsc{Prec}^-(\hat{X})$$

4.2. Results on Instructed Track

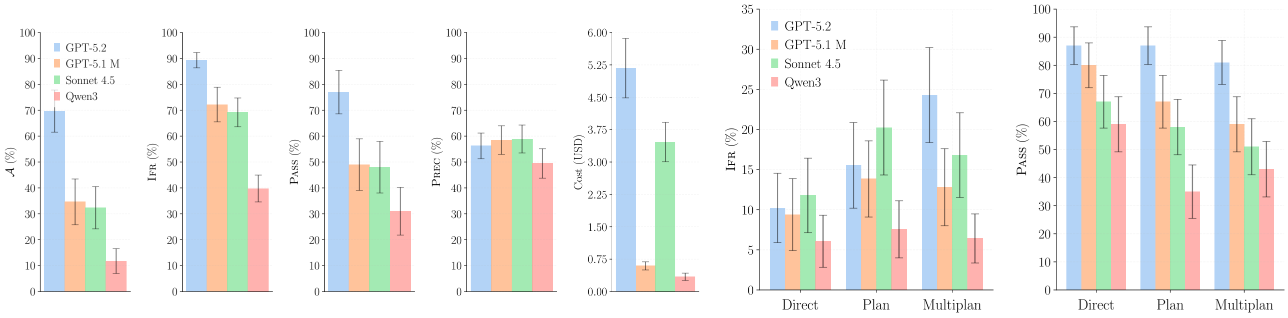
Even in the instructed track, where agents are provided with a detailed blueprint specifying the exact refactoring transformations required, notable performance gaps emerge among frontier systems. Unless otherwise specified, all metrics reported in this section and in §4.3 concern averages across instances.

SOTA agents have major differences As can be seen in Figure 7a, GPT-5.2 significantly outperforms all other agents on alignment $\mathcal{A}$, achieving an average $\mathcal{A}$ of 69.6%. In contrast, SONNET 4.5 and GPT-5.1 M achieve only 32.4% and 34.6%, respectively. The open-source QWEN3 lags substantially behind at 11.8%.

Notably, all of GPT-5.2, GPT-5.1 M and SONNET 4.5 have high IFR of 89.3%, 72.2% and 69.2%. However, the $\mathcal{A}$ of the latter two is significantly lower than for GPT-5.2, since their PASS is roughly 30 percentage-points below GPT-5.2's 76.0% at 47.0% and 43.0%, respectively. Thus, while all frontier agents are able to follow detailed and large scale changes, there are large gaps in maintaining code correctness. In our case studies we find that GPT-5.2 achieves higher PASS rates by applying targeted, individual patches, wheras SONNET 4.5 runs insensitive search and replace commands. While one might suspect that propensity to validate outputs with the test suite might have a strong effect, GPT-5.2 actually runs tests less frequently than SONNET 4.5.

Overall high precision Overall, all models achieve high precision scores. With SONNET 4.5, GPT-5.1 M and GPT-5.2 achieving 58.9%, 58.5%, and 56.2%, respectively. All of these scores effectively cluster around human precision at 57.5%. This suggests that when the refactoring intent is clearly specified, frontier agents can limit their changes to the relevant scope without introducing substantial unrelated modifications.

Thorough refactorings are expensive The higher alignment of GPT-5.2 is resource-intensive. To manage agentic execution costs, the agents are subject to a \$11 monetary



(a) Results on the Instruct Track

(b) Results on the Open Track

Figure 7. (Left) On the instruct track, GPT-5.2 substantially outperforms other models across IFR, PASS, and $\mathcal{A}$. (Right) In the Open Track, the propose-then-implement strategy (Plan) and oracle selection (Oracle Multiplan) improve IFR for GPT-5.2. PREC remains at most 21.0% across the open track, indicating substantial unrelated changes.

budget per task; runs exceeding this limit are terminated and evaluated with available output. GPT-5.2 hit this limit in 14 instances, compared to only 3 instances for SONNET 4.5. The mean cost for GPT-5.2 in instructed mode is \$5.17, compared to \$3.46 for SONNET 4.5 and \$0.59 for GPT-5.1 M. This is due to the aforementioned targeted patches by GPT-5.2, while SONNET 4.5 relies on cheaper search and replace commands (e.g. `sed -i`).

4.3. Results on the Open Track

Achieving high IFR on the open track is significantly harder. This is because agents are only provided with a high-level focus area, and need to recover human architectural choices. The maximal achieved alignment score $\mathcal{A}$ over all models is 7.7% in Direct Inference mode, as can be seen in Figure 7b.

Models need planning for large refactorings When the models first propose a plan before implementing it (Plan Mode), there is a significant increase in $\mathcal{A}$. For GPT-5.2, the $\mathcal{A}$ nearly doubles from 7.7% to 14.1%. On average, the scores increase by a total of almost 3%, corresponding to a more than 50% relative increase. This suggests that explicit reasoning before editing helps in recovering large-scale human refactoring intents. Case studies (Appendix C.1) reveal that planning tends to yield repository-wide transformations that reflect high-level architectural thinking. The corresponding patches are approximately 2 to 3 times larger (measured by $|L_{\hat{X}}|$) and achieve up to a 72% relative increase in IFR. The broader scope can come at the cost of up to 24% lower PASS rates (Direct vs. Plan). QWEN3 exhibits the most significant decline, whereas GPT-5.2 sees no decline. Details across all models and modes are presented in Appendix Table 4.

Suggestions generally include human choices Among all generated proposals, the models more frequently suggest at least one human suggestion. This is visible in Oracle Multiplan, where they generate several proposals, and the closest plan to the desired refactoring is chosen by an LLM with oracle knowledge. In this setting GPT-5.2 reaches an alignment score $\mathcal{A}$ of 19.4%. Interestingly, SONNET 4.5 does not improve in this setting, with its score dropping slightly from 10.2% in Plan mode to 9.7%. This might indicate that, when tasked to generate several possible refactorings, the model shifts into a different, less aligned distribution of refactoring plans.

4.4. Failure Modes in Open Track

To answer what the open track actually tests, we investigated a sample of solutions predicted by GPT-5.2 and SONNET 4.5 and compared the golden solution with their solutions. Some observed issues concern (a) fixating on salient, obvious subjects, e.g., a typo in the name of a class, and completely ignoring the broader focus area, (b) resorting to lazy, hacky workarounds, e.g., creating a compatibility shim to bypass updating hundreds of import statements, (c) relying on destructive, repository-wide replace-all string commands, leaving the codebase in a broken and inconsistent state, and (d) working on entirely different aspects of the system than required. While we cannot predict what a realistic upper bound for human alignment is on a refactoring, we observe that even the best current models have major issues on the refactoring choices they make. We include examples of LLM-assisted manual analysis on such cases in Appendix C.

How much guessing is needed? When generating the static analysis rules, we did not show the agent the modified code. This was done so that the agent would not overfit rules to the actual code edit. However, some of the rules assert that elements with certain names should appear in the codebase. We believe this could lead to harder rules in the open track. To quantify how much of a concern this is, we compared the IFR^{+} and IFR^{-} across modes. In Direct mode, GPT-5.2 has IFR^{+} of 8.9% and IFR^{-} of 10.3%. In Plan mode, IFR^{+} is 9.9% and IFR^{-} is 17.2%, while in Multiplan mode IFR^{+} and IFR^{-} rise to 15.7% and 26.7%, respectively. This indicates that while some rules may involve guessing elements such as names, such problems likely affect approximately half of the additive rules. We provide detailed results in the Appendix in Table 3.

5. Related Work

Prior benchmarks have measured more basic refactorings conducted by LLMs or coding agents, which we detail in this section.

Single-File Refactoring Benchmarks A number of benchmarks measure changes only on single-file edits. RefactorMirror (Liu et al., 2025) is a Java-only benchmark that checks whether LLM refactoring suggestions align with human preferences. They observe generally low alignment with human preferences, but do not verify the proposal by having the LLM implement the refactoring. Cordeiro et al. (2024) evaluate the ability of Starcoder to decrease generic code complexity metrics on 30 Java tasks. Rejiba et al. (2025) measure code complexity metrics when tasking LLMs to refactor C++ and C code. The aider team introduced a benchmark for Python code that evaluates function-level refactorings (Aider-AI, 2024). Finally, Cassano et al. (2023) present instructions of varying degree of detail to LLMs to refactor Python code on a function-level and measure whether generated code matches manually specified syntactic properties.

Multi-File Refactoring Benchmarks RefactorBench (Gautam et al., 2025) analyses GPT-4o refactoring suggestions and implementations on a 100-instance Python dataset. On average, they edit 4.3 files and test instruction following with syntactic validation against expected structures. SWE-refactor (Xu et al., 2026) is a Java-only benchmark for simple, few-file and line refactorings MiniCode (Kovacic et al., 2025) measure whether LLMs can successfully discover shared code and extract it into a library function, thus limit themselves to a fairly simple task. In our benchmark, the average number of edited files 91.52, significantly higher than prior works. Further, our work covers 6 distinct languages. Thus, CodeTaste is a significantly more challenging benchmark, even in the instruct track.

Repository-level Code Benchmarks Spearheaded by Jimenez et al. (2024), evaluating coding agents on the autonomous resolution of real-word repository-level tasks quickly became the gold standard for assessing their capabilities. While initial work focuses on issue resolution (Jimenez et al., 2024), follow up work proposed benchmarks on feature addition (Li et al., 2025; Du et al., 2025), unit test generation (Mündler et al., 2024), function generation (Liang et al., 2025), code performance (He et al., 2025), and security (Chen et al., 2025).

Methods to improve refactoring Pomian et al. (2024) evaluate whether models can decide when to extract code into external methods. Zhang et al. (2024) evaluate the function-level capability of models at rewriting Python code into idiomatic expressions. Shirafuji et al. (2023) similarly prompt GPT-3.5 with few-shot prompts to evaluate its refactoring capabilities. All of these methods are limited to very specific refactorings, while our benchmark covers a wide range of potential refactorings.

6. Conclusion

We introduced CodeTaste, a benchmark for evaluating refactoring agents on real, multi-file changes mined from open-source repositories. Our pipeline pairs each refactoring with executable environments and OpenGrep-based static rules that verify the intended transformation beyond test passing. Our benchmark features two complementary tracks: an instructed track that measures diligence in the execution of detailed refactorings, and an open track that measures alignment with human refactoring choices under underspecification.

Our results demonstrate that when provided with explicit refactoring instructions, current agents achieve relatively strong, yet unsaturated, performance. GPT-5.2 stands out as the most capable model, achieving a compound alignment score $\mathcal{A}$ of 69.6%, leaving room for improvement across the board. Alignment collapses when only a focus area is provided. Propose-then-implement strategies and oracle selection over multiple plans increase alignment by up to 2.44-fold, yet remain far from human choices. These findings highlight that autonomous judgment about *what* to refactor remains a key bottleneck. We hope CodeTaste serves as both a rigorous evaluation target and a preference signal for training agents that can preserve long-term code quality, and we see expanding the dataset, strengthening rule and test coverage, and learning better proposal mechanisms as promising directions for future work.

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here.

References

Aider-AI. Aider-AI/refactor-benchmark, 2024. URL https://github.com/Aider-AI/refactor-benchmark. Aider's refactoring benchmark exercises based on popular python repos. Apache-2.0 license. Last updated 2024-10-10.

Anthropic. Claude sonnet 4.5 system card, September 2025. URL https://www-cdn.anthropic.com/963373e433e489a87a10c823c52a0a013e9172dd.pdf.

Badertdinov, I., Golubev, A., Nekrashevich, M., Shevtsov, A., Karasik, S., Andriushchenko, A., Trofimova, M., Litvintseva, D., and Yangel, B. Swe-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents, 2025. URL https://arxiv.org/abs/2505.20411.

Carlini, N. Building a C compiler with a team of parallel Claudes. https://www.anthropic.com/engineering/building-c-compiler, 2026. Anthropic Engineering Blog.

Cassano, F., Li, L., Sethi, A., Shinn, N., Brennan-Jones, A., Lozhkov, A., Anderson, C. J., and Guha, A. Can it edit? evaluating the ability of large language models to follow code editing instructions. *CoRR*, 2023.

Chen, J., Huang, H., Lyu, Y., An, J., Shi, J., Yang, C., Zhang, T., Tian, H., Li, Y., Li, Z., Zhou, X., Hu, X., and Lo, D. Secureagentbench: Benchmarking secure code generation under realistic vulnerability scenarios, 2025. URL https://arxiv.org/abs/2509.22097.

Code Claude Docs. Claude code overview, 2025. URL https://code.claude.com/docs/en/overview.

Cordeiro, J., Noei, S., and Zou, Y. An empirical study on the code refactoring capability of large language models. *CoRR*, 2024.

donbarbos. Most popular github repositories (projects). URL https://www.kaggle.com/datasets/donbarbos/github-repos. Kaggle shows a relative timestamp: "Updated 2 years ago" (as of access date).

Du, Y., Cai, Y., Zhou, Y., Wang, C., Qian, Y., Pang, X., Liu, Q., Hu, Y., and Chen, S. Swe-dev: Evaluating and training autonomous feature-driven software development, 2025. URL https://arxiv.org/abs/2505.16975.

Gautam, D., Garg, S., Jang, J., Sundaresan, N., and Zilouchian, R. Refactorbench: Evaluating stateful reasoning in language agents through code. In Yue, Y., Garg, A., Peng, N., Sha, F., and Yu, R. (eds.), *International Conference on Representation Learning*, volume 2025, pp. 43131–43162, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/6b44ee74539ea77d6a0d50d468724371-Paper-Conference.pdf.

GitHub. Github repos public dataset. https://console.cloud.google.com/marketplace/details/github/github-repos, 2016. Public dataset on Google BigQuery; part of the BigQuery Public Datasets program.

Grigorik, I. Gh archive, 2011. URL https://www.gharchive.org/.

He, X., Liu, Q., Du, M., Yan, L., Fan, Z., Huang, Y., Yuan, Z., and Ma, Z. Swe-perf: Can language models optimize code performance on real-world repositories?, 2025. URL https://arxiv.org/abs/2507.12415.

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. R. SWE-bench: Can Language Models Resolve Real-world Github Issues? In *ICLR*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Kovacic, Z., Lee, C., Chiu, J. T., Zhao, W., and Ellis, K. Refactoring codebases through library design. *CoRR*, 2025.

Li, W., Zhang, X., Guo, Z., Mao, S., Luo, W., Peng, G., Huang, Y., Wang, H., and Li, S. FEA-bench: A benchmark for evaluating repository-level code generation for feature implementation. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 17160–17176, Vienna, Austria, July 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.acl-long.839. URL https://aclanthology.org/2025.acl-long.839/.

Liang, S., Hu, Y., Jiang, N., and Tan, L. Can language models replace programmers for coding? repocod says 'not yet', 2025. URL https://arxiv.org/abs/2410.21647.

Liu, B., Jiang, Y., Zhang, Y., Niu, N., Li, G., and Liu, H. Exploring the potential of general purpose llms in automated software refactoring: an empirical study. *Autom. Softw. Eng.*, (1), 2025.

Mündler, N., Müller, M. N., He, J., and Vechev, M. SWT-Bench: Testing and Validating Real-world Bug-fixes with Code Agents. In *NeurIPS*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/94f093b41fc2666376fb1f667fe282f3-Abstract-Conference.html.

Mürtz, C. and Müller, M. N. Agents in the wild - dashboard. https://insights.logicstar.ai, 2025. URL https://doi.org/10.5281/zenodo.15846865. Interactive web dashboard. Code available at https://github.com/logic-star-ai/insights.

OpenAI. Introducing upgrades to codex, 2025a. URL https://openai.com/index/introducing-upgrades-to-codex/.

OpenAI. openai/codex: Lightweight coding agent that runs in your terminal, 2025b. URL https://github.com/openai/codex. GitHub repository under Apache-2.0 license, a local coding agent from OpenAI.

Orlanski, G., Roy, D., Yun, A., Shin, C., Gu, A., Ge, A., Adila, D., Albarghouthi, A., and Sala, F. Slop-CodeBench: Measuring Code Erosion Under Iterative Specification Refinement, 2024. URL https://github.com/SprocketLab/slop-code-bench.

Perry, D. E. and Wolf, A. L. Foundations for the study of software architecture. *ACM SIGSOFT Softw. Eng. Notes*, (4), 1992.

Pomian, D., Bellur, A., Dilhara, M., Kurbatova, Z., Bogomolov, E., Bryksin, T., and Dig, D. Next-generation refactoring: Combining LLM insights and IDE capabilities for extract method. In *ICSME*, 2024.

QwenLM. Qwenlm/qwen3-coder: Qwen3-coder is the code version of qwen3, the large language model series developed by qwen team, alibaba cloud, 2025. URL https://github.com/QwenLM/Qwen3-Coder.

Rejiba, Z., Eidenbenz, R., and Borkhoche, J. Research: A tool and workflow for benchmarking llm-based code improvements. In *SIGSOFT FSE Companion*, 2025.

Sarkar, S. K. Ai agents, productivity, and higher-order thinking: Early evidence from software development. *Available at SSRN 5713646*, 2025.

Shirafuji, A., Oda, Y., Suzuki, J., Morishita, M., and Watanobe, Y. Refactoring programs using large language models with few-shot examples. In *APSEC*, 2023.

Singh, A., Fry, A., Perelman, A., Tart, A., Ganesh, A., El-Kishky, A., McLaughlin, A., Low, A., Ostrow, A., Ananthram, A., Nathan, A., Luo, A., Helyar, A., Madry, A., Efremov, A., Spyra, A., Baker-Whitcomb, A., Beutel, A., Karpenko, A., Makelov, A., Neitz, A., Wei, A., Barr, A., Kirchmeyer, A., and Ivanov, A. Openai gpt-5 system card, 2025. URL https://arxiv.org/abs/2601.03267.

Wilson Lin. wilsonzlin/fastrender, 2026. URL https://github.com/wilsonzlin/fastrender. Scaling long-running autonomous coding.

Xu, Y., Yang, J., Tse-Hsun, and Chen. Swe-refactor: A repository-level benchmark for real-world llm-based code refactoring, 2026. URL https://arxiv.org/abs/2602.03712.

Yang, A., Li, A., Yang, B., Zhang, B., Hui, B., Zheng, B., Yu, B., Gao, C., Huang, C., Lv, C., Zheng, C., Liu, D., Zhou, F., Huang, F., Hu, F., Ge, H., Wei, H., Lin, H., Tang, J., Yang, J., Tu, J., Zhang, J., Yang, J., Yang, J., Zhou, J., Zhou, J., Lin, J., Dang, K., Bao, K., Yang, K., Yu, L., Deng, L., Li, M., Xue, M., Li, M., Zhang, P., Wang, P., Zhu, Q., Men, R., Gao, R., Liu, S., Luo, S., Li, T., Tang, T., Yin, W., Ren, X., Wang, X., Zhang, X., Ren, X., Fan, Y., Su, Y., Zhang, Y., Zhang, Y., Wan, Y., Liu, Y., Wang, Z., Cui, Z., Zhang, Z., Zhou, Z., and Qiu, Z. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

Zhang, Z., Xing, Z., Ren, X., Lu, Q., and Xu, X. Refactoring to pythonic idioms: A hybrid knowledge-driven approach leveraging large language models. *Proc. ACM Softw. Eng.*, (FSE), 2024.

A. Experimental Details, Ablations and Case Study

In this section, we provide additional details about the implementation, hyperparameters, datasets, and so on.

A.1. Data curation pipeline details

We obtain a list of commits for further processing using a three-stage pipeline.

Stage 1: Querying and Keyword Filtering We filter out all repositories with less than 90 stars. Among those, we search for commits that edit at least 45 files and feature specific keywords. Concretely, let K^+ and K^- denote the sets of keywords indicative of refactoring and non-refactoring commits, respectively. We further distinguish between strong ($K^{\pm}_{\text{strong}}$) and weak ($K^{\pm}_{\text{weak}}$) keywords, which form subsets of K^+ and K^-. For instance, `deduplicate` $\in K^+_{\text{strong}}$, `reorganize` $\in K^+_{\text{weak}}$, `fix` $\in K^-_{\text{strong}}$, and `lint` $\in K^-_{\text{strong}}$.

Commit messages M of candidate commits must satisfy:

$$\text{Count}(M, K^-_{\text{strong}}) = 0 \quad \text{and} \quad \frac{\text{Count}(M, K^+)}{\text{Count}(M, K^-_{\text{weak}})} \geq 2$$

We then obtain the top 5000 commits sorted lexicographically descending by (Count(M, K^+_{strong}), Count(M, K^+_{weak}), #stars, #files changed).

Stage 2: Prefiltering We remove all repositories and commits that are not accessible anymore, i.e., have been deleted or made private. To exclude pure documentation changes, we require a minimum of 25 edits in files that have a file extension of typical programming languages, such as `.py` or `.cpp`. Via manual inspection we determine that we need to remove extreme outliers, which include for example entire virtual environments. We retain only those commits that satisfy:

$$0.5p_{10} \leq \text{changes} \leq 2p_{90}$$

where changes refers to the number of added or removed lines, and the number of created, deleted, modified, or moved files. Finally, to exclude highly repetitive and simple search-and-replace refactorings, we sort the results by the ratio of $\frac{|\text{diff}_c|}{|\text{diff}|}$, where $|\text{diff}_c|$ is the size of the Gzip level 6 compressed diff (excluding file context) in bytes.

Stage 3: LLM Scoring In the final step we use a small LLM (CLAUDE 4.5 HAIKU) to analyze the commit messages and code diffs. The prompts for this step can be found in Figures 9 and 10.

First, the LLM is prompted to score the commit message across several dimensions: a score for being a project-wide change S_{large}, a score for behavior-preserving refactoring S_{refactor}, and a message cohesion score S_{cohesion}. The model also estimates a length penalty P_{length} for multi-topic messages, a triviality penalty P_{trivial} for simple edits, and a penalty for general uncertainty P_{unc}. We retain the top 1,120 candidates based on the overall message score S_{msg} calculated as:

$$\begin{aligned} S_{\text{msg}} = {} & (0.35 S_{\text{large}} + 0.45 S_{\text{refactor}} + 0.2 S_{\text{cohesion}}) \\ & - (0.2 P_{\text{length}} + 0.2 P_{\text{trivial}} + 0.2 P_{\text{unc}}) \end{aligned}$$

Next the LLM is prompted to analyse the code diff, focusing on the refactoring likelihood S_{refactor} and the estimated complexity $S_{\text{complexity}}$ of the change. It also identifies penalties for mixed content P_{mixed}, evidence of behavior-altering logic P_{risky}, signs of automation P_{auto}, uncertainty P_{unc}, and the presence of trivial changes P_{trivial}. The diff-specific score S_{diff} is defined by:

$$\begin{aligned} S_{\text{diff}} = {} & (0.8 S_{\text{refactor}} + 0.2 S_{\text{complexity}}) - 1.5(0.4 P_{\text{mixed}} \\ & + 0.2 P_{\text{risky}} + 0.1 P_{\text{unc}} + 0.15 P_{\text{trivial}} + 0.15 P_{\text{auto}}) \end{aligned}$$

We retain the top 160 candidates based on a combined score S_{total} that prioritizes the diff analysis over the commit message:

$$S_{\text{total}} = \frac{S_{\text{msg}} + 3 S_{\text{diff}}}{4}$$

Running the pipeline This pipeline is run four times, once each on the main table of the GitHub Activity Data dataset (GitHub, 2016) and the three main tables for 2023, 2024 and 2025 respectively in the GitHub Archive (Grigorik, 2011), filtered to `PullRequestEvent` for merged pull requests. We obtain the count of "stars", an indication of popularity, for each repository from donbarbos. The final result are $4 \times 160 = 640$ candidates.

A.2. Details of the execution environment agent

We use a polyglot container based on Ubuntu 24.04 with pre-configured toolchains and version managers for common language ecosystems. The base image includes: (i) Python 3.8–3.11 managed via `uv`, (ii) Node.js 22.12.0 (default) managed via Node Version Manager (NVM), (iii) Go v1.23.4 and a Rust toolchain (`cargo`), (iv) `OpenJDK`, and (v) `C/C++` via GCC/Clang.

The build environment agent, SONNET 4.5 in the CLAUDE CODE harness, is restricted to \$5 cost and 90 minutes time. To ensure portability of the benchmark, we limit the size of the writable storage layer, after all setup is completed, to 5GB. Further, we enforce a 15-minute time limit on running the test suite. This container is then committed.

In the runtime setup, we concretely remove the git commit history and remote origins to prevent leakage of golden commits.

A.3. Details of the rule agent

In this section, we detail the specific commands that the rule discovery agent can use.

- `BashExplore`$_R$`(cmd)`: Executes a set of read-only shell commands in the base codebase R to identify relevant code patterns (e.g., `ls`, `rg`).
- `Register(`$\Gamma_{\text{candidates}}$`)`: Submits candidate rules in `yaml` syntax. Without loss of generality, we show the registration of additive rules only.

  Registration: Every valid additive rule is registered:

  $$\Gamma^+ = \Gamma^+ \cup \{\gamma \in \Gamma_{\text{candidates}} \mid M(\gamma, R \circ X^*) > 0 \\ \wedge M(\gamma, R) = 0 \quad \}$$

  To prevent redundant rules, candidates must be sufficiently distinct from previously registered rules; we require pairwise Jaccard overlap of matched lines to be at most 0.95.

  Feedback: For each $\gamma \in \Gamma_{\text{candidates}}$, the agent receives the match counts $M(\gamma, R)$ and $M(\gamma, R \circ X^*)$, as well as rejection reasons (if applicable). In the case of execution failures (e.g., `SyntaxError` or `OpengrepError`), the agent is forwarded parts of error message for refining the rules in the next iteration. For highly discriminative but invalid rules where $M(\gamma, R \circ X^*) > 10 \times M(\gamma, R)$, the agent is provided with a witness $\omega \in \Omega(\gamma, R)$ to facilitate adding exemptions.[1]
- `Finish()`: Signals completion and returns the valid registered rule set.

---

[1] In many real-world refactorings there are edge cases such as unadjusted examples that can prevent a rule from reaching $M(\gamma, R) = 0$. Providing witnesses for these matches allows the agent to construct more general patterns by exempting specific outliers.

A.4. Complete list of used repositories

We provide a complete list of the 87 repositories included in CODETASTE in Table 2. We have checked the license for all repositories and confirm they are compliant to be used in CODETASTE.

Table 2. Complete list of used repositories.

99designs/gqlgen	Automattic/wp-calypso	DataDog/datadog-agent
NationalSecurityAgency/ghidra	PrestaShop/PrestaShop	TanStack/router
actualbudget/actual	ansible/ansible	antvis/G6
apache/flink	apache/hadoop	apache/nuttx
apache/pinot	apache/pulsar	apache/shardingsphere
apache/shenyu	aquasecurity/tracee	aws/aws-cli
badges/shields	bentoml/BentoML	bevyengine/bevy
boa-dev/boa	burn-rs/burn	bytebase/bytebase
chaos-mesh/chaos-mesh	chroma-core/chroma	cilium/cilium
clientIO/joint	conan-io/conan	containerd/nerdctl
cosmos/cosmos-sdk	deepset-ai/haystack	elastic/eui
etcd-io/etcd	ethereum/go-ethereum	excalidraw/excalidraw
fabricjs/fabric.js	getsentry/sentry	ggerganov/llama.cpp
go-gitea/gitea	gohugoio/hugo	golang/go
golangci/golangci-lint	google/gvisor	google/tink
gopasspw/gopass	gradle/gradle	graphhopper/graphhopper
hashicorp/consul	highcharts/highcharts	hrydgard/ppsspp
huggingface/transformers	immich-app/immich	influxdata/influxdb
istio/istio	jerryjliu/llama_index	juju/juju
kedro-org/kedro	knative/serving	koajs/koa
kubevirt/kubevirt	lima-vm/lima	loft-sh/vcluster
mne-tools/mne-python	mockito/mockito	mongodb/mongo-go-driver
monkeytypegame/monkeytype	n8n-io/n8n	netty/netty
nhost/nhost	opensearch-project/OpenSearch	prisma/prisma
pulumi/pulumi	pyvista/pyvista	remarkablemark/html-react-parser
ruffle-rs/ruffle	rust-lang/rust	saleor/saleor
sanity-io/sanity	scikit-learn/scikit-learn	syncthing/syncthing
temporalio/temporal	tinymce/tinymce	twosigma/beakerx
vitessio/vitess	wader/fq	wikimedia/apps-android-wikipedia

A.5. Complete Data for Instruction Following and Test Passing

We provide detailed results on the instruction following, alignment score and test pass rate for the different models and settings in Tables 3 and 4. For completeness, we also report $\mathcal{A}^+$ and $\mathcal{A}^-$, which are defined with respect to only the $\textsc{Ifr}^+$ and $\textsc{Ifr}^-$, respectively, below.

$$\mathcal{A}^+(\hat{X}) = \textsc{Pass}(\hat{X}) \times \textsc{Ifr}^+(\hat{X})$$
$$\mathcal{A}^-(\hat{X}) = \textsc{Pass}(\hat{X}) \times \textsc{Ifr}^-(\hat{X})$$

Table 3. Full experimental results across all tracks and models in percents on $\textsc{Ifr}^+$ (additive instruction-following rate), $\textsc{Ifr}^-$ (reductive instruction-following rate), and PASS (test pass rate).

		GPT-5.2			GPT-5.1 M			SONNET 4.5			QWEN3		
		$\textsc{Ifr}^+$	$\textsc{Ifr}^-$	PASS	$\textsc{Ifr}^+$	$\textsc{Ifr}^-$	PASS	$\textsc{Ifr}^+$	$\textsc{Ifr}^-$	PASS	$\textsc{Ifr}^+$	$\textsc{Ifr}^-$	PASS
Instructed Track		90.1	88.7	76.0	76.9	69.0	47.0	78.8	62.6	43.0	55.6	30.6	30.0
Open Track	Direct	8.9	10.3	87.0	7.1	9.7	80.0	9.6	12.1	67.0	3.4	6.7	59.0
Open Track	Plan	9.9	17.2	87.0	8.1	15.1	67.0	15.4	21.7	58.0	3.4	8.8	35.0
Open Track	Multiplan	15.7	26.7	81.0	7.3	14.2	59.0	14.5	18.0	51.0	5.1	7.0	43.0

Table 4. Full experimental results across all tracks and models for PASS, $\mathcal{A}$, IFR in percents with 95 % confidence intervals.

		PASS (%)	$\mathcal{A}$ (%)	$\mathcal{A}^{+}$ (%)	$\mathcal{A}^{-}$ (%)	IFR (%)	IFR^{+} (%)	IFR^{-} (%)
GPT-5.2								
Instructed		76.0 ± 8.5	69.6 ± 8.1	68.8 ± 8.3	69.6 ± 8.4	89.3 ± 2.9	90.1 ± 3.5	88.7 ± 4.0
Open	Direct	87.0 ± 6.7	7.7 ± 3.9	5.8 ± 3.1	7.9 ± 4.2	10.2 ± 4.3	8.9 ± 4.0	10.3 ± 4.7
Open	Plan	87.0 ± 6.7	14.1 ± 5.3	7.5 ± 4.0	16.1 ± 6.0	15.5 ± 5.3	9.9 ± 4.6	17.2 ± 6.0
Open	Multiplan	81.0 ± 7.8	19.4 ± 5.7	10.7 ± 4.0	22.2 ± 6.7	24.3 ± 5.9	15.7 ± 4.9	26.7 ± 6.8
GPT-5.1 Codex Mini								
Instructed		47.0 ± 10.0	34.6 ± 8.8	34.7 ± 8.9	34.7 ± 9.0	72.2 ± 6.7	76.9 ± 6.1	69.0 ± 7.6
Open	Direct	80.0 ± 8.0	6.9 ± 4.0	4.8 ± 3.6	7.5 ± 4.3	9.4 ± 4.5	7.1 ± 4.1	9.7 ± 4.7
Open	Plan	67.0 ± 9.4	9.2 ± 4.2	4.7 ± 3.2	10.5 ± 4.8	13.8 ± 4.7	8.1 ± 3.9	15.1 ± 5.4
Open	Multiplan	59.0 ± 9.8	4.0 ± 2.7	1.9 ± 1.9	4.7 ± 3.1	12.8 ± 4.8	7.3 ± 4.1	14.2 ± 5.4
Claude Sonnet 4.5								
Instructed		43.0 ± 9.9	32.4 ± 8.1	34.2 ± 8.6	31.4 ± 8.4	69.2 ± 5.6	78.8 ± 4.8	62.6 ± 7.3
Open	Direct	67.0 ± 9.4	6.1 ± 3.5	4.4 ± 3.1	6.4 ± 3.8	11.8 ± 4.6	9.6 ± 4.4	12.1 ± 5.0
Open	Plan	58.0 ± 9.8	10.2 ± 4.6	6.1 ± 3.9	11.7 ± 5.4	20.2 ± 5.9	15.4 ± 5.5	21.7 ± 6.7
Open	Multiplan	51.0 ± 10.0	9.7 ± 4.7	6.5 ± 4.0	10.7 ± 5.3	16.8 ± 5.3	14.5 ± 5.3	18.0 ± 6.1
Qwen3								
Instructed		30.0 ± 9.1	11.8 ± 4.8	17.2 ± 6.1	8.7 ± 4.6	39.8 ± 5.2	55.6 ± 5.6	30.6 ± 6.1
Open	Direct	59.0 ± 9.8	1.9 ± 2.1	0.6 ± 0.7	2.2 ± 2.4	6.1 ± 3.2	3.4 ± 2.7	6.7 ± 3.7
Open	Plan	35.0 ± 9.5	2.0 ± 2.0	0.8 ± 0.7	2.2 ± 2.3	7.6 ± 3.6	3.4 ± 2.5	8.8 ± 4.4
Open	Multiplan	43.0 ± 9.9	2.0 ± 2.1	1.0 ± 1.3	2.6 ± 2.6	6.4 ± 3.1	5.1 ± 3.2	7.0 ± 3.6
Golden Patch		100.0	100.0	100.0	100.0	100.0	100.0	100.0
Base Commit		100.0	0.0	0.0	0.0	0.0	0.0	0.0

Table 5. Full experimental results across all tracks and models for PREC in percents with 95 % confidence intervals. And average LoC edited $|L_{\hat{X}}|$ in agent patches.

		PREC (%)	PREC^{+} (%)	PREC^{-} (%)	$\lvert L_{\hat{X}} \rvert$	$\lvert L^{+}_{\hat{X}} \rvert$	$\lvert L^{-}_{\hat{X}} \rvert$
GPT-5.2							
Instructed		56.2 ± 5.0	59.3 ± 5.7	52.8 ± 6.0	2432.9	1163.3	1269.6
Open	Direct	18.0 ± 5.4	10.7 ± 5.0	26.4 ± 7.3	767.7	367.5	400.2
Open	Plan	15.5 ± 4.9	8.9 ± 4.4	24.6 ± 6.6	2279.7	892.4	1387.3
Open	Multiplan	21.0 ± 5.0	12.5 ± 4.5	34.6 ± 7.2	2624.3	1694.9	929.4
GPT-5.1 Codex Mini							
Instructed		58.5 ± 5.5	60.3 ± 5.9	55.5 ± 6.4	1307.2	606.8	700.4
Open	Direct	15.9 ± 5.2	7.8 ± 4.5	27.3 ± 7.8	488.9	211.7	277.2
Open	Plan	14.2 ± 4.2	5.3 ± 3.0	26.0 ± 7.1	1042.7	558.5	484.1
Open	Multiplan	15.1 ± 4.7	7.5 ± 4.3	26.1 ± 7.0	666.8	356.1	310.7
Claude Sonnet 4.5							
Instructed		58.9 ± 5.4	57.5 ± 5.9	57.1 ± 6.4	2100.6	1306.6	794.0
Open	Direct	12.5 ± 4.3	8.0 ± 4.3	23.9 ± 7.1	1535.2	822.9	712.4
Open	Plan	14.1 ± 4.2	9.1 ± 4.2	26.0 ± 6.6	3278.1	1470.1	1808.0
Open	Multiplan	12.6 ± 4.3	8.7 ± 4.3	25.1 ± 6.5	3826.1	2993.2	832.9
Qwen3							
Instructed		49.4 ± 5.7	47.7 ± 6.2	48.5 ± 6.9	1610.3	1071.9	538.4
Open	Direct	5.8 ± 3.1	4.2 ± 3.3	11.1 ± 5.5	2067.2	227.2	1840.0
Open	Plan	10.2 ± 4.1	5.3 ± 3.7	17.7 ± 6.0	1141.3	487.3	654.0
Open	Multiplan	9.1 ± 3.8	5.1 ± 3.6	21.6 ± 6.9	787.1	359.8	427.3
Golden Patch		57.5	61.5	51.9	2434.2	1126.8	1307.4
Base Commit		0.0	0.0	0.0	0.0	0.0	0.0

B. Examples

In this section, we provide an example of a rule that is used in the evaluation of CODETASTE. The provided code snippets are for illustrative purposes only.

```
opengrep scan -f rule.yml example.js
```

Example Rule with Symbolic Propagation

rule.yml

```yaml
rules:
  - id: eui-theme-colors-access
    message: Detected euiTheme.colors property access
    severity: INFO
    languages:
      - typescript
      - javascript
    pattern: $THEME.colors.$PROP
    options:
      symbolic_propagation: true

```

example.js

```javascript
import { euiTheme } from '@elastic/eui';

function ThemeComponent() {
  const myTheme = euiTheme;
  const themeColors = myTheme.colors;
  const primaryColor = themeColors.primary; // match (symbolic propagation)
  return <div style={{ color: primaryColor }}>Hello World</div>;
}

function DirectThemeComponent() {
  const primaryColor = euiTheme.colors.primary; // match (direct)
  return <div style={{ color: primaryColor }}>Hello World</div>;
}

```

Figure 8. The rule matches both direct access (`euiTheme.colors.primary`) and aliased access through intermediate variables (e.g., `myTheme` and `themeColors`).

C. Open Track Case Studies

In this section, we present case studies comparing agent outputs against the ground truth refactorings. For each case, we use an `claude-code` with OPUS 4.6 to assist us in parsing the generated patches obtained from GPT-5.2and SONNET 4.5, and their respective evaluation artifacts and help us compare them to the ground truth X^*. We then manually review the resulting report, verify the claims and present the following takeaways. The prompt used in this part can be found in Figure 19.

C.1. Mockito

The refactoring restructures Mockito's flat `org.mockito.internal` package into five sub-packages (`creation`, `invocation`, `state`, `stubbing`, `verification`), renames several core classes and methods (e.g., `VerifyingMode` to `OngoingVerifyingMode`, `andThrows` to `andThrow`), and updates all imports and cross-references accordingly. The changes span approximately 2,000 lines across the core Mockito source. The precise commit is `mockito/mockito@2f7bf91d`.

C.1.1. OPEN TRACK

The task provided to the agents is: `Improve internal organization and naming`.

Direct Mode. In direct mode, none of the agents proceeds to perform a refactoring that resembles the scope of X^*, instead they are attracted to fixing tiny, surface-level nitpicks. They preserve and validate functional correctness.

SONNET 4.5 renamed typo-containing hamcrest matcher classes and removed `I` prefixes from interface names (`IArgumentMatcher` to `ArgumentMatcher`) and successfully runs tests. It achieved an IFR of 0.76%, by coincidentally removing the `IAnswer` file.

GPT-5.2 fixed the same typos using a backwards-compatible deprecated-wrapper approach, replaced a wildcard import, cleaned up some parameter names, and successfully runs tests. It performed none of the structural package moves or renames contained in the ground-truth. It achieved an IFR of 0.0%.

Plan Mode. Both agents resort to a similar set of superficial edits. Both successfully run tests and builds.

SONNET 4.5 focuses on renaming typo-containing classes. Destructive file renames created duplicate test classes. No structural changes were made. It achieved an IFR of 0.0%.

GPT-5.2 focuses on fixing typos in class names and improving overall naming. No structural changes were made. It achieved an IFR of 0.76% by coincidentally renaming the field `mocksToBeVerifiedInSequence` to `mocksToVerifyInOrder`.

Multiplan Mode. The edits are no longer purely superficial, but address fundamental code quality concerns. Each of the plans created by SONNET 4.5 addresses a basket of issues, resulting in incomplete edits. The plans proposed by GPT-5.2 are more focused resulting in a consistent and correct patch.

SONNET 4.5 achieved an IFR of 6.9%, the agent addressed state package extraction (1/5 subpackages), performed no type/method renames, yet introduced new unnecessary abstractions (e.g. StateManager interface). The introduction of a partial field update results in missing field errors causing tests and build to fail. The agent is not able to recover from this inconsistent state.

GPT-5.2 achieved an IFR of 29.0%. The agent successfully reorganized all 5 target subpackages with correct import/package updates across 60+ files, but did not attempt any of the type or method renames that account for the remaining 50% of positive rules. The agent successfully builds the updated codebase.

C.1.2. INSTRUCTED TRACK

Both agents address the specifications provided in the detailed task description, resulting in a perfect or near perfect IFR of 99% and 100% for SONNET 4.5 and GPT-5.2, respectively. GPT-5.2 uses targeted patches for its edits, resulting in a successful build. SONNET 4.5 makes extensive use of risky replace all `sed -i` commands on all `.java` files in the `test` directory. This results in an inconsistent state and build errors.

C.2. AWS CLI

The refactoring migrates the `aws-cli` test suite from the third-party `mock==1.3.0` package to Python's built-in `unittest.mock`, updating `awscli/testutils.py` and `tests/__init__.py` as centralized re-export points and rewriting imports across 178 test files to use project-idiomatic `from tests import mock` or `from awscli.testutils import mock` patterns.

The precise commit is `aws/aws-cli@20315462`.

C.2.1. OPEN TRACK

The task provided to the agents is: `Update test mocking dependencies`.

Direct Mode. SONNET 4.5 achieved an IFR of 14.3% by correctly updating the two infrastructure files and 144 of 178 test files, but used `from unittest import mock` directly in each file instead of the ground-truth's project-idiomatic `from tests import mock` / `from awscli.testutils import mock` convention. The generated patch $\hat{X}$ is not complete, however the IFR also punishes that the agent did not use the `from tests import mock` convention, which is a bit harsh.

GPT-5.2 achieved an IFR of 2.9% by updating only the two infrastructure files and creating a `sys.modules` compatibility shim (`tests/mock.py`) to make old `import mock` statements work transparently, rather than actually migrating any test file imports; the near-zero score accurately reflects that the agent bypassed the refactoring entirely, leaving all 178 test files unchanged with their old-style import patterns intact.

Plan Mode. Both agents perform a similar set of edits as in the reference refactoring X^*. SONNET 4.5 validates its edits by running tests, GPT-5.2 only validates the syntax using `python -m compileall`. The IFRs achieved are 56% and 74% for GPT-5.2 and SONNET 4.5, respectively. The two resulting codebases pass our test suite.

The behavior in **Multiplan Mode** and **Instructed Track** is almost identical. Both agents achieve 74%, the non-perfect score stems from using `from awscli.testutils import mock` instead of `from tests import mock`.

C.3. gqlgen

The refactoring modernizes the entire gqlgen Go codebase by replacing all occurrences of `interface{}` with Go 1.18's built-in `any` alias, a purely mechanical refactoring affecting files across library code, templates, tests, and examples, with no behavioral changes. The precise commit is `99designs/gqlgen@d5c9f896`.

C.3.1. OPEN TRACK

The task provided to the agents is `Modernize type declarations across the codebase`.

Direct Mode. Both agents perform the refactoring

SONNET 4.5 achieved an IFR of 93.3%, successfully replacing `interface{}` with `any` across 193 files and passing all 17 additive rules, but left behind `interface{}` in a small number of locations (6 out of 73 reductive rule violations), including some variable declarations, unmarshal function parameters, and function return types.

GPT-5.2 achieved a perfect IFR of 100.0%, comprehensively replacing every `interface{}` occurrence with `any` across 231 files with zero remaining legacy patterns detected, fully accomplishing the refactoring.

Plan Mode. SONNET 4.5 achieved IFR of 63.3% by replacing `interface{}` with `any` in core runtime, client, config, handler, and plugin source files. It ran `go test` repeatedly, but did not run `go generate` to propagate changes to generated code, leaving residues in generated files. GPT-5.2 achieved IFR of 78.9% by replacing `interface{}` with `any` across source files and running `go generate ./...` to propagate template changes to generated code. It never ran tests and exhausted its \$11 budget before completion. The resulting patch $\hat{X}$ contains genuine gaps in hand-written files (e.g., `complexity/` package, various Unmarshal signatures) that the agent did not reach before budget exhaustion.

Multiplan Mode. SONNET 4.5 achieved an IFR of 100% by performing a comprehensive repository-wide `sed`-based replacement of `interface{}` to `any` across 232 files, and running tests successfully multiple times.

GPT-5.2 achieved an IFR of 100% via a `perl`-based global replacement followed by `go generate` across 231 files, and preserved functional correctness. It claims to have run tests, but actually did not.

C.3.2. INSTRUCTED TRACK

SONNET 4.5 achieved IFR of 100% by performing a comprehensive `sed -i`-based global replacement of `interface{}` with `any` across all `.go` filesIt adds a linter rule and verifies correctness via `go build` and `go test`.

GPT-5.2achieved IFR of 96.7% using `perl` and `gofmt -r` to correctly handle most replacements. Three residual occurrences of `interface{}` in parameter/field declarations remain however. These violations were not captured by `gofmt -r`. The test suite is passing even though the agent never actually runs `go test` or `go build`.

C.4. Bytebase

The refactoring standardizes Vue.js route constant naming across 51 files by adding a `_ROUTE_` infix to project route constants, extracting workspace setting route names from hardcoded strings into exported constants, refactoring sidebar components for route-name-based navigation, and creating a reusable `WebhookTypeIcon` component. The precise commit is `bytebase/bytebase@04ac644e`.

C.4.1. OPEN TRACK

Direct Mode. The task provided to the agents is `Refactor routing system`.

SONNET 4.5 refactored the Go backend server routing (middleware extraction, service registry pattern) rather than the intended Vue.js frontend route constants, achieving an IFR of 0.0%.

GPT-5.2 removed the legacy `routeSlug` method and `RouterSlug` type from the frontend router store, which is completely orthogonal to the constant naming standardization addressed in the human refactoring. It achieved an IFR of 0.0%.

The plans proposed in **Plan Mode** and **Multiplan Mode** are larger in scope, but remain orthogonal to the frontend refactoring X^*. The IFR achieved for both agents in both modes remains at 0.0%.

C.4.2. INSTRUCTED TRACK

SONNET 4.5 achieved IFR 100% by successfully applying the core route-constant renaming and setting-route centralization across 43 files. This is 8 files less compared to the golden patch X^*. The IFR of 100% is inflated due to binary rule satisfaction constraints as defined in §4.1. The agent uses broad `sed -i` commands resulting in much lower cost compared to GPT-5.2.

GPT-5.2 achieved IFR 100% by applying the refactoring more broadly than X^*, touching 87 files (vs 51 in X^*), adding route constants in files beyond the scope of X^*. The agent proceeded using many targeted patches until finally hitting the budget constraint.

D. Prompts

In this section, we detail all prompts used for the respective models and tasks. The prompts include: (i) scoring prompts used in the data creation and filtering pipeline (Figures 9 and 10), (ii) task generation prompts for the instructed and open tracks (Figures 13 and 14), and (iii) prompts used during evaluation for planning, execution, and oracle plan selection (Figures 11, 12 and 15–18).

SONNET 4.5

Commit Message Scoring

System

```
You are a precise refactoring change-description classifier. We want to find out whether the change corresponds to a
large-scale refactor that exceeds simple replace-all, but does not introduce new features, bug fixes, etc.
Given either a commit message or a PR title+body, output ONLY an XML payload with numeric scores in [0,1].
Do not include any prose outside XML.

Definitions:
- large_scale_refactor_score: Likelihood this describes a sweeping, project-wide refactor across many files or modules.
  Indicators: global rename/codemod, API migration, package/module reorg, bulk moves, mass formatting, automated changes.
  Anti-indicators: single-file tweak, small fix, narrow scope.
- true_refactor_score: Likelihood changes are behavior-preserving refactoring (no new features, no bug fixes).
- message_cohesion_score: How focused and single-purpose the message is.
  High: one clear goal. Low: laundry list (many unrelated bullets, "misc", "various", repeated "and/also").
- length_penalty: Penalty for messages suggesting many in-between or multi-topic commits.
  High penalty for very long, multi-issue messages; near 0 for short, focused messages.
- trivial_score: Likelihood that the commit involves only trivial changes, e.g. generated by a single sed command or only involves file moves/renames.
- uncertainty: An estimate of how uncertain your assessment is, based on ambiguity or lack of detail.

Clamp to [0,1].

Output format (XML only):
<classification>
  <large_scale_refactor_score>...</large_scale_refactor_score>
  <true_refactor_score>...</true_refactor_score>
  <message_cohesion_score>...</message_cohesion_score>
  <length_penalty>...</length_penalty>
  <uncertainty>...</uncertainty>
  <trivial_score>...</trivial_score>
  <reasoning>One-paragraph justification</reasoning>
</classification>
```

User

```
Refactor environment, exception handling and jumping in VM
~~Depends on #3053~~
Fixes #2424
This PR refactors environment handling and jumping with break, continues, etc in the VM. Currently the opcodes are very complex, they dynamically search for the
       environments, for this we also need to keep a lot of state in the `CallFrame`, this has a performance penalty as well as being hard to implement
      optimizers (since it has to mirror the state, so it does not change behaviour).
Since this the bytecompiler has full knowledge of where it needs to jump, how many environments to pop, etc. This PR aims to move that opcode logic to
      bytecompiler, generating opcodes that are "dumber"/simple, hence easier to optimize ( using #3037 ).
~~This still needs a lot of work!~~
It changes the following:
- Removes opcodes: `FinallyStart`, `FinallyEnd`, `Break`, `Continue`, `LoopStart`, `LoopEnd`, `LabelledStart`, `LabelledEnd`, `TryStart`, `TryEnd`, `
      IteratorLoopStart`,`GeneratorResumeReturn`, `GeneratorSetReturn`, `GeneratorJumpOnResumeKind`
[...]
```

Assistant

```
<classification>
  <large_scale_refactor_score>0.88</large_scale_refactor_score>
  <true_refactor_score>0.82</true_refactor_score>
  <message_cohesion_score>0.72</message_cohesion_score>
  <length_penalty>0.12</length_penalty>
  <trivial_score>0.08</trivial_score>
  <uncertainty>0.12</uncertainty>
  <reasoning>This is a large-scale architectural refactor of the VM's environment and control-flow handling. The message clearly describes sweeping changes
      across multiple subsystems: removing 10 opcodes, adding 6 new ones, eliminating the EnvEntry type and env entry stack, removing/modifying CallFrame fields
      , and restructuring logic from the VM to the bytecompiler. The refactor is explicitly motivated by simplification for optimization, not new features. The
      message is focused on a single coherent theme despite covering multiple technical details, with a reasonably detailed but not excessive structure. This is
       unambiguously a true refactor (behavior-preserving rewrite) rather than a trivial rename or bug fix.</reasoning>
</classification>
```

Figure 9. Commit message scoring prompt used as a late-stage filter in the data scraping funnel.

SONNET 4.5

Diff Scoring

System

```
Your task is to identify whether the unified diff provided is a true, useful large-scale refactoring commit.
Given a unified diff (git format), output ONLY an XML payload with numeric scores in [0,1].
Do not include any prose outside XML.
Definitions and guidance:
- refactor_likelihood: Likelihood changes are behavior-preserving refactoring (rename/move, code reorganization, function extraction/inlining, variable/method
      renames, import/namespace changes, signature reshaping, mass formatting).
- mixed_change_penalty: Penalty if the diff mixes refactors with features/bugfixes or risky changes.
- risky_change_score: Evidence of behavior-altering edits (new logic, feature flags, new APIs).
- automated_pattern_score: Signs that refactoring results from automated tools, linters or similar.
- uncertainty: Reflects ambiguity, lack of context, or insufficient signal in the diff.
- complexity: Give an estimate of how complex this refactor is. (0=trivial, 0.5=moderate, 1=extremely complex)
- trivial_score: Indicates the presence of trivial changes (e.g., whitespace, comments).

Output format (XML only):
<classification>
  <refactor_likelihood>...</refactor_likelihood>
  <mixed_change_penalty>...</mixed_change_penalty>
  <risky_change_score>...</risky_change_score>
  <automated_pattern_score>...</automated_pattern_score>
  <uncertainty>...</uncertainty>
  <reasoning>One-paragraph justification</reasoning>
  <complexity>...</complexity>
  <trivial_score>...</trivial_score>
</classification>
```

User

```
diff --git a/core/engine/src/builtins/eval/mod.rs b/core/engine/src/builtins/eval/mod.rs
index 3e6c905e5fd..55b38f31405 100644
--- a/core/engine/src/builtins/eval/mod.rs
+++ b/core/engine/src/builtins/eval/mod.rs
@@ -20,7 +20,7 @@ use crate::{
     string::StaticJsStrings,
-    vm::{CallFrame, CallFrameFlags, Constant, Registers},
+    vm::{CallFrame, CallFrameFlags, Constant},
@@ -333,8 +333,7 @@ impl Eval {
-        let register_count = context.vm.frame().code_block().register_count;
-        let record = context.run(&mut Registers::new(register_count as usize));
+        let record = context.run();
         context.vm.pop_frame();
[...]
```

Assistant

```
<classification>
  <refactor_likelihood>0.92</refactor_likelihood>
  <mixed_change_penalty>0.05</mixed_change_penalty>
  <risky_change_score>0.25</risky_change_score>
  <automated_pattern_score>0.75</automated_pattern_score>
  <uncertainty>0.08</uncertainty>
  <reasoning>This is a large-scale, systematic refactoring that consolidates VM register and stack management. The `Registers` struct is eliminated and its
      functionality is merged into a new `Stack` struct. All ~50+ opcode handler functions are consistently updated to remove the `&mut Registers` parameter and
       access registers through `context.vm` instead. CallFrame methods are refactored to return indices rather than values. The changes are behavior-preserving
       reorganization with very consistent patterns across all files, showing evidence of automated tooling supplemented by manual coordination. While the
      structural changes are deep and touch core VM systems (register access, promise capabilities, async generators), the refactoring appears carefully
      coordinated with no obvious feature additions or bug fixes mixed in.</reasoning>
  <complexity>0.85</complexity>
  <trivial_score>0.12</trivial_score>
</classification>
```

Figure 10. Diff scoring prompt used to rank candidate commits by refactoring quality.

SONNET 4.5

Environment Setup with `claude-code`

System

```
## Your Environment

You operate in a containerized, non-interactive **polyglot development environment** based on **Ubuntu 24.04**.

### **Core Runtimes**

* **Python:** Managed by `uv` (versions **3.8, 3.9, 3.10, 3.11**, see `uv python list`).
* **Node.js:** **v22.12.0** (via NVM `nvm`) with **TypeScript**, `ts-node`, and `vercel`.
* **Go:** **v1.23.4** (Global toolchain).
* **Rust:** Full toolchain (Cargo/Rustup) in `/opt/rust`.
* **.NET:** **SDK 8.0** (LTS).
* **C/C++:** GCC, Clang, CMake, and `build-essential`.
* **Java:** OpenJDK (`default-jdk`).

### **Pre-installed Tools**

* **Testing:** **Playwright** (with Chromium and system deps).
* **System:** `git`, `curl`, `wget`, `vim`, `sudo`, and build libraries (SSL, ffi, sqlite).

### **Environment Specs**

* **User:** `benchmarker` (non-root with passwordless `sudo`).
* **Repository Directory:** `/testbed`. `/testbed` is wiped on exit.
* **Key Paths:** Tools are pre-configured in `$PATH` (Node, Rust, .NET, Go, uv).
## Task

Configure the development environment and validation scripts for the repository located in `/testbed/`.

**Constraints:**

1. **EXPLORE:** Analyze the `/testbed/` directory to identify the primary programming language, required runtime versions (e.g., via version files or manifests), the preferred
      package manager, and the testing framework used.
2. **DEPS:** Identify and install necessary system-level dependencies using `sudo` (non-interactive) and all project-level dependencies. Ensure any external binaries or drivers
      required by the test suite (e.g., browser engines, compilers, or database headers) are installed immediately. Anything installed inside `/testbed/` will be wiped on exit!
3. **INTEGRITY:** NEVER modify files in /testbed/ directly, any changes you perform will be wiped after you exit! `/scripts/setup_shell.sh` can setup /testbed directory, however it
      should NOT modify versioned files in `/testbed/`, i.e. it should only modify files or folders that are explicitly ignored by the version control system (e.g., build artifacts,
       dependency directories, cache). `git status` must show no changes.
4. **SCRIPTS:**
* **Create `/scripts/setup_system.sh`:**
    Executed with `sudo` prior to running the tests, this script performs runtime system configuration (e.g., starting database services, Redis, or configuring system limits).
    It should **not** install packages. If no system services are required, create a script that exits 0.
* **Create `/scripts/setup_shell.sh`:**
    When sourced, this script configures the shell environment for the project **and** to run tests:
        - activate virtual environment if necessary.
        - install local project dependencies and linters.
        - set up all environment variables.
        - selects the correct runtime versions.
        - Project and dependency installations that change files in `/testbed/` MUST be performed in this script.
    It must **NOT** require `sudo`. It must be idempotent (safe to run multiple times) and avoid redundant installations.
* **Create `/scripts/run_tests`:** This script must be self-contained and execute the test suite (or a relatively large and representative subset that finishes in up to 15 minutes).
       It will be invoked as follows: `git clean -xdff && sudo /scripts/setup_system.sh && source /scripts/setup_shell.sh && /scripts/run_tests`.
    `/scripts/run_tests` must NOT setup the environment, environment variables, shell or similar, it should rely on the previous scripts having been run before. `/scripts/run_tests`
       must only run and parse tests. It should be invoked like : `/scripts/setup_shell.sh && /scripts/run_tests` to run tests in an already setup shell. `/scripts/setup_shell.sh`
      sets up the environment and environment variables.
* **Portability:** These scripts **must remain functional** even if `/testbed/` is checked out to THE previous commit (in other words: `HEAD~1`). I.e. they should work on HEAD and
      HEAD~1 without any modifications.
5. **OUTPUT:** The `/scripts/run_tests` script must output exactly one JSON line as its final `stdout` message:
`{"passed": int, "failed": int, "skipped": int, "total": int}`.
Where the int value corresponds to the number of tests that passed, failed, were skipped, and the total number of representative tests run.
You may need to write a wrapper or use a custom test reporter to ensure this exact format is captured. Test results must be **deterministic** and reflect actual test outcomes.
6. **VERIFICATION:** Always execute `source /scripts/setup_shell.sh && /scripts/run_tests` in a single shell command (in an environment that is clean -> `git clean -xdff` and has
      system services running `/scripts/setup_system.sh`) on both the current commit and `HEAD~1`. Confirm the JSON outputs accurately reflect the test results.
Your shell doesn't persist environment across shell commands.

**Note:** Multiple runtimes (Python, Node, Rust, .NET, Java, GO) and their respective version managers are pre-installed. Any global configurations or runtime versions you install
      will be preserved across the session.
**Note:** You run in a non-interactive terminal; Proceed with these actions without asking for further permission. Take action and perform the installation and changes.

## Final Notes
Once you have completed the task, briefly write a concise summary of the testing setup in `/scripts/SUMMARY.md` of the form:
# Summary
...
## System Dependencies
...
## PROJECT Environment
...
## Testing Framework
...
## Additional Notes
... (e.g. any obstacles or misconfigurations of the environment you encountered)
```

Figure 11. Setup Prompt for Environment Configuration

boa-dev/boa@ac9eb4bc

Example Instance Scripts

setup_system.sh

```
#!/bin/bash
set -e

# No system services required for this Rust project.
exit 0
```

setup_shell.sh

```
#!/bin/bash
set -e

cd /testbed

if ! command -v cargo-nextest &> /dev/null; then
    cargo install cargo-nextest --locked
fi

cargo build --all-targets --profile ci --features annex-b,intl_bundled,experimental,embedded_lz4
cargo test --no-run --profile ci --features annex-b,intl_bundled,experimental,embedded_lz4
```

run_tests

```
#!/bin/bash
set -e

cd /testbed

cargo nextest run --profile ci --cargo-profile ci --features annex-b,intl_bundled,experimental,embedded_lz4 2>&1 | tee /tmp/test_output.txt || true

SUMMARY=$(grep -E "^[[:space:]]+Summary" /tmp/test_output.txt | tail -1 || echo "")
TOTAL=$(echo "$SUMMARY"  | grep -oP '[0-9]+(?= tests run)' || echo "0")
PASSED=$(echo "$SUMMARY" | grep -oP '[0-9]+(?= passed)'    || echo "0")
FAILED=$(echo "$SUMMARY" | grep -oP '[0-9]+(?= failed)'    || echo "0")
SKIPPED=$(echo "$SUMMARY"| grep -oP '[0-9]+(?= skipped)'   || echo "0")

printf '{"passed": %s, "failed": %s, "skipped": %s, "total": %s}\n' "$PASSED" "$FAILED" "$SKIPPED" "$TOTAL"
```

Figure 12. Excerpt of the standardized scripts generated during the setup phase for the `boa-dev/boa` instance (base commit `ac9eb4bc`).

SONNET 4.5

Task Generation (Instructed Track)

System

~~~~
Generate a refactoring task description for the commit described by the user. It should be formatted like a GitHub issue.
Restrict yourself to the scope of this commit, do not talk about Follow-Up Tasks, Future or Past Work.
Only output the issue description, nothing else. Do not include any explanations.

[...]

Output Format should be well structured:
Title
-----
<title>
Summary
-------
<summary>
Why
---
<why>
[...]
~~~~

User

~~~~
<repo_name>boa-dev/boa</repo_name>
<commit_message>Refactor registers to use the stack (#4263)
* Refactor registers to use the stack
* Fix import in doc comment</commit_message>
<pull_requests><pull_request>
<pr_title>Refactor registers to use the stack</pr_title>
<pr_body>This Pull Request changes the following:
- Refactor the vm stack into a dedicated struct and implement stack access in functions that are more readable.
- Remove the `Registers` data structure and store registers on the stack.
</pr_body></pull_request>
</pull_requests>
<commit_diff>diff --git a/core/engine/src/builtins/eval/mod.rs b/core/engine/src/builtins/eval/mod.rs
index 3e6c9..55b38 100644
--- a/core/engine/src/builtins/eval/mod.rs
+++ b/core/engine/src/builtins/eval/mod.rs
@@ -22,3 +22,3 @@ use crate::{
     string::StaticJsStrings,
-    vm::{CallFrame, CallFrameFlags, Constant, Registers},
+    vm::{CallFrame, CallFrameFlags, Constant},
     Context, JsArgs, JsResult, JsString, JsValue, SpannedSourceText,
@@ -335,4 +335,3 @@ impl Eval {
-        let register_count = context.vm.frame().code_block().register_count;
-        let record = context.run(&mut Registers::new(register_count as usize));
+        let record = context.run();
[...]</commit_diff>
~~~~

Assistant

~~~~
# Title
-----
Refactor registers to use the stack

# Summary
-------
Refactor VM execution to store registers directly on the stack instead of maintaining a separate `Registers` data structure. Consolidate stack and register
      management into a single `Stack` type with improved API and readability.

# Why
---
- Eliminate duplication between stack and registers storage
- Improve memory efficiency by using single unified structure
- Better encapsulation with dedicated `Stack` type and semantic accessor methods
- Clearer separation between frame pointer, arguments, and register spaces

# Changes
---

**Removed `Registers` struct:**
- Deleted standalone `Registers` data structure (~50 lines)
- Removed `registers.push_function()` / `registers.pop_function()` calls
- No more `Registers::new()` or `registers.clone_current_frame()`

**New `Stack` struct:**
- Encapsulates `Vec<JsValue>` with ~200 lines of accessor methods
- Methods: `get_this()`, `get_function()`, `get_arguments()`, `get_register()`, `set_register()`, ...
- Calling convention helpers: `calling_convention_pop_arguments()`, `calling_convention_push_arguments()`, ...
- Frame management: `truncate_to_frame()`, `split_off_frame()`
- Special register accessors: `get_promise_capability()`, `async_generator_object()`, ...

[...]
```
~~~~

Figure 13. Task description generation prompt used to synthesize an issue-style refactoring task from commit associated metadata.

SONNET 4.5

Open Track Task Generation

System

```
Turn this description into a generic, high-level, abstract instruction that **doesn't describe what** has to be done. It should leave out design decisions.
      Limit to one EXTREMELY CONCISE and high-level sentence, phrased as human-like instruction. We want close to NO detail in the instruction, i.e. a very
      GENERIC instruction, that only slightly nudges in the right direction. Output must be plain text.
Example: Refactor ABC and XYZ API
```

User

```
Title
-----
Refactor registers to use the stack
Summary
-------
[Remainder of Instructed Track Task Description]
```

Assistant

```
Consolidate VM registers and stack
```

Figure 14. Prompt used to derive the open-track instruction by abstracting away actionable detail from an instructed task description.

ANY AGENT

Open Track: Plan Prompt

~~~
Conduct IN-DEPTH EXPLORATION and ANALYSIS of the codebase.
Refine the following task description into a CONCRETE and ACTIONABLE refactoring plan. Make ALL the design choices. DO NOT propose broad, multi-stage changes.
You operate completely AUTONOMOUSLY in this sandboxed environment. DO NOT ASK FOR CLARIFICATIONS.
The final plan must be in markdown format:
```
# Title
<Summary>
## Changes
...
## Why
...
```
You MUST PLACE the final plan in this file: '/output/refactoring_plan.md'.

Task:
[Task Description]
~~~

Figure 15. Prompt used to generate a single concrete refactoring plan from an open-track instruction.

ANY AGENT

Open Track: Oracle Multiplan Prompt

~~~
Conduct IN-DEPTH EXPLORATION and ANALYSIS of the codebase.
Generate 5 DISTINCT and DIFFERENT refactoring plans for the following task. There can be partial overlap.
You operate completely AUTONOMOUSLY in this sandboxed environment. DO NOT ASK FOR CLARIFICATIONS.

For each approach, create a COMPLETE, CONCRETE and ACTIONABLE refactoring plan. Make ALL the design choices. DO NOT propose broad, multi-stage changes.
Each plan must be in markdown format:
```
# Title
<Summary>
## Changes
...
## Why
...
```

- '/output/refactoring_plans/refactoring_plan0.md'
- '/output/refactoring_plans/refactoring_plan1.md'
- '/output/refactoring_plans/refactoring_plan2.md'
- '/output/refactoring_plans/refactoring_plan3.md'
- '/output/refactoring_plans/refactoring_plan4.md'

Task:
[Task Description]
~~~

Figure 16. Prompt used to generate multiple distinct candidate refactoring plans from an open-track instruction.

SONNET 4.5

Open Track: Oracle Plan Selection Prompt

User

```
You are evaluating multiple refactoring plans for a software engineering task.

Your goal is to select the plan that BEST matches the original task description in terms of similarity / overlap.

Below is the ORIGINAL TASK DESCRIPTION:

---
[ORIGINAL TASK DESCRIPTION]
---

Now, here are FIVE CANDIDATE REFACTORING PLANS:

PLAN 0:
---
[PLAN 0]
---

PLAN 1:
---
[PLAN 1]
---

PLAN 2:
---
[PLAN 2]
---

PLAN 3:
---
[PLAN 3]
---

PLAN 4:
---
[PLAN 4]
---

Based on your analysis, which plan (0, 1, 2, 3, or 4) BEST matches the original task description?

Provide your reasoning briefly, then state your final selection clearly as:
SELECTED: [plan number]

Your selection:
```

Figure 17. Prompt used by the oracle judge to select the candidate plan that best matches the original refactoring description.

ANY AGENT Inference Prompt

```
Perform the task described below in it's ENTIRETY. You operate completely AUTONOMOUSLY in a sandboxed environment. DO NOT ASK FOR CLARIFICATIONS. You must EDIT the codebase DIRECTLY to complete the task. DO NOT create reports, plans or similar files.

[Task Description]
```

Figure 18. Default Prompt used to run inference on an agent.

OPUS 4.6 `claude-code` Judge Prompt

~~~
You should create a report comparing the outputs of the two agents (`claude-code-v2.0.76-sonnet45`, `codex-v0.77.0-gpt-5.2`) to the ground-truth refactoring (`golden`).
Ignore .venv, .node_modules, .cargo-home and binary file edits in your report.
Give an assesment of whether the score regarding IFR is justified based on the evidence. I.e. does it give an accurate reflection on how much of the true reference refactoring the agent achieved.
Additionally, look at the agents performance regarding functional correctness by investigating the test results.
Connect the analysis to the inference.out to answer obvious questions, such as did the agent try to run tests or build?
Talking or thinking about running a command does not count as evidence. Only the actual commands and their outputs count.

Explore all files in this folder / subfolders to find evidence for your report, but only edit the `case_study_report.md` file.
Place the report inside the current directory. It should be a markdown file named `case_study_report.md`.
```md
[Evidence, Reasoning, Claims]

## Dense Summary

The refactoring [... 1-2 sentence summary about what the ground-truth refactoring achieves.]

claude-code-v2.0.76-sonnet45: [1-2 sentences about what the agent achieved, what IFR score is achieved (why and whether this is reflective), and did the agent preserve functional correctness? Why / why not? Also say what wasn't achieved if relevant.]
codex-v0.77.0-gpt-5.2: [1-2 sentences about what the agent achieved, what IFR score is achieved (why and whether this is reflective), and did the agent preserve functional correctness? Why / why not? Also say what wasn't achieved if relevant.]
```
~~~

Figure 19. Prompt used to run `claude-code` to assist evaluation of the different tracks in the case studies.