RefactorBench-JS: Evaluating LLM Agents on Behavior-Preserving Code Decomposition

Daniel Chen[1]

[1]Anything, Inc.

Abstract

We introduce RefactorBench-JS, an open benchmark for evaluating AI coding agents on behavior-preserving software maintenance tasks. The benchmark measures whether LLM agents can decompose large JavaScript and React files into smaller modules without changing observable behavior. It consists of 123 scored fixtures—spanning algorithmic logic, data modules, utility/API logic, web UI, and mobile UI—each paired with a hidden unit test suite: hidden from the agent during evaluation, then released publicly for auditability. An agent's output is scored by executing the hidden tests against the refactored filesystem, directly measuring whether behavior was preserved. This design is grounded in a simple principle: *behavioral preservation is a functional property, and functional properties are best verified by functional tests*—not by code similarity metrics, static analysis, or LLM-as-judge.

We demonstrate RefactorBench-JS by evaluating Anthropic Claude and Google Gemini models across two tool configurations. The benchmark reveals model-dependent tool effects, large gaps between agent-reported success and actual behavioral preservation, and distinct failure modes across model families. We report hidden-test pass rates, calibration, operational metadata, and a failure taxonomy. These findings informed model and tool selection in a production refactoring system whose observed success rate improved from approximately 65% to above 97% amid multiple engineering changes; this production result is observational, not a controlled causal estimate. RefactorBench-JS is publicly available at https://github.com/Create-Inc/refactor-bench.

Keywords: AI agent evaluation, coding agents, benchmark, code refactoring, large language models, tool augmentation

1 Introduction

AI coding agents are increasingly expected to carry out multi-step software engineering tasks, but evaluating their reliability remains difficult when success depends on preserving program behavior rather than producing plausible code. Automated refactoring—restructuring code without changing its observable behavior—is a particularly sharp test case: it requires an agent to edit, split, and reconnect code while keeping the user-visible system unchanged. Existing approaches rely on metrics that are either too weak (compilation success, file count) or misaligned with the task (code similarity, LLM-as-judge). These metrics can show encouraging numbers even when the refactored code is functionally broken.

We encountered this problem firsthand while building an automated refactoring subagent for a commercial AI application builder. The subagent decomposes large React files into smaller modules to prevent downstream failures in speculative editing. Early evaluation using compilation success and file count metrics showed steady improvement—yet users reported broken applications after automatic refactoring. We could not distinguish a refactoring that compiled but broke behavior from one that was genuinely correct.

The solution was conceptually simple: *behavioral preservation is a functional property, and functional properties are best verified by functional tests.* We constructed a set of files paired with hidden unit test suites, invisible to the refactoring agent, that exercise each file's observable behavior. After refactoring, the hidden tests are executed against the refactored filesystem. If they pass, behavior was preserved; if they fail, it was not.

This paper introduces RefactorBench-JS, the public release of this evaluation suite. Our contributions are:

1. **AI-agent benchmark release**: RefactorBench-JS is an open benchmark of 123 scored JavaScript/React fixtures with hidden test suites for evaluating behavior preservation in LLM-based refactoring agents, released at `https://github.com/Create-Inc/refactor-bench`.

2. **A behavior-grounded evaluation design**: we adapt the established software-engineering principle of functional regression testing to LLM

refactoring agents, and contrast executable hidden tests with code similarity metrics, static analysis, and LLM-as-judge.

3. **Agent performance baselines, calibration, and failure analysis**: we evaluate seven frontier and fast models across two tool configurations and report hidden-test pass rates, agent self-evaluation calibration, operational cost/latency metadata, and failure buckets.

2 Motivation: Refactoring in AI Code Generation

To ground RefactorBench-JS in a concrete use case, we briefly describe the production system that motivated its creation.

2.1 The Speculative Editing Bottleneck

Modern AI application builders use a *base agent* to generate code edits, which are applied to existing files via *speculative editing* (also called *fast apply*). This merge step has a practical constraint: when a file exceeds approximately 16,000 tokens, speculative editing begins to fail, producing malformed merges. To manage file size, a *refactoring subagent* is invoked automatically when a file crosses 8,000 tokens. Its task is to decompose the file into smaller modules while preserving behavior.

2.2 Why Evaluation Was the Bottleneck

The refactoring subagent is implemented as a multi-turn tool-calling agent. It was straightforward to equip it with tools (file writing, build checking, test execution, a termination signal) and iterate on prompts. The bottleneck was *evaluation*: without a reliable signal for behavioral preservation, every design decision—model selection, tool configuration, prompt changes—was made in the dark. Compilation success showed a system that appeared to work. Users showed a system that did not.

Critically, subagent quality propagates: the worse the refactoring subagent performs, the more likely the base agent must spend additional turns—using a more expensive frontier model—diagnosing and repairing the damage. A cheap but unreliable subagent is a false economy. This dynamic made reliable evaluation not just desirable but essential.

This experience motivated the construction of the hidden-test evaluation suite, which we now release as RefactorBench-JS so that other teams building refactoring agents can avoid the same trap.

3 The RefactorBench-JS Benchmark

3.1 Design Principles

RefactorBench-JS is built on three design principles:

1. **Functional evaluation.** The primary metric is whether hidden unit tests pass after refactoring, directly measuring behavioral preservation. Structural metrics (file count, compilation, code similarity) are recorded but are secondary.

2. **Hidden holdout tests.** The test suites are never provided to the refactoring agent. The agent may write and execute its own tests using a test runner tool, but these are distinct from the holdout suite. This separation serves two purposes. First, it prevents the agent from optimizing directly for the evaluation criteria. Second, the hidden tests are designed to cover behaviors that a self-testing agent is unlikely to verify on its own: multi-step user interaction flows (register $\rightarrow$ search $\rightarrow$ chat $\rightarrow$ send), browser API side effects (localStorage persistence across theme toggles), and cross-validation of multiple algorithm implementations against each other. In practice, agents' self-written tests tend to be shallow smoke tests ("component renders without crashing"), whereas the hidden suites exercise deeper behavioral invariants.

3. **Domain diversity.** The corpus spans pure algorithms, data modules, utility/API logic, React web components with complex state management, and React Native mobile components at varying complexity levels.

3.2 Corpus

The benchmark consists of 123 scored fixtures, each in its own directory with a consistent structure:

```
refactoring/data/<name>/
  eval.config.json              # app metadata (name, type)
  refactoring_eval.config.json
  src/
  *.test.js
```

Table 1 summarizes the corpus by platform metadata recorded in the repository.

The corpus deliberately spans several complexity regimes:

Table 1: RefactorBench-JS corpus summary. LOC = physical line count in the target file, counted by newline splitting. The full fixture inventory is the repository's `refactoring/data/` directory.

Platform	Fixtures	LOC range	Representative behaviors tested
Web / JavaScript	90	48–4,218	Algorithms, data modules, APIs, rendering, events, browser APIs
Mobile / React Native	33	227–2,597	Navigation, gestures, native APIs, modals, mobile state flows
Total	**123**	**48–4,218**	

Algorithm files. These test whether the agent preserves pure functional behavior—input–output contracts, algorithmic correctness, and data structure invariants—during decomposition. The hidden tests exercise edge cases (empty inputs, single elements, large inputs) and cross-validate multiple implementations of the same algorithm (e.g., multiple Fibonacci implementations that must produce identical results).

Data module files. These test preservation of exported data structures, configuration objects, and generator functions. Hidden tests verify data integrity: required fields, unique IDs, value type constraints, and cross-referential consistency.

Utility / API files. These bridge the gap between pure algorithms and full UI components, testing server-side API routes, command processors, custom hooks, and business logic handlers. Hidden tests mock external dependencies (database clients, auth, fetch) and verify request handling, validation rules, error responses, and state transitions.

React web component files. These test preservation of UI rendering, event handling, state management, and browser API interactions. Larger fixtures include authentication flows, real-time messaging, theme toggling with localStorage persistence, and developer badge systems. The hidden tests exercise full user flows: register $\rightarrow$ search $\rightarrow$ select chat $\rightarrow$ send message.

React Native mobile files. These test mobile-specific patterns: navigation, gesture handling, native module interactions, and platform-specific layouts. Files span domains including sports scheduling, financial services, grocery shopping, games, and social features.

3.3 Hidden Test Suite Design

Hidden test suites were written or reviewed to follow these guidelines:

- **Test observable behavior.** Assert on rendered output, return values, state transitions, and side effects—not internal structure. This makes the tests invariant under correct refactoring.
- **Cover edge cases.** Include empty inputs, boundary conditions, error handling, and multi-step interaction flows.
- **Use stable selectors.** For React components, test by text content, role, and semantic structure rather than CSS classes or internal component names, which refactoring may change.

For algorithm files, tests use Jest-style assertions on function return values. For React components, tests use React Testing Library with user-event for interaction simulation.

3.4 Scoring

RefactorBench-JS produces five scores per refactoring attempt:

Table 2: Scoring metrics.

Score	Type	Description
Passes Tests	Binary	Do the hidden holdout tests pass against the refactored filesystem? *Primary metric.*
Agent Reported Success	Binary	Did the agent signal success via its termination tool?
Non-Triviality	Binary	Did the refactoring produce new files (not a no-op)?
Files Compiled	$[0, 1]$	$1 -$ (build errors/total files).
Cost	\$	Total LLM cost in dollars.

The *Passes Tests* and *Agent Reported Success* scores can diverge: an agent may signal success while the hidden tests reveal regressions. This divergence

rate measures the agent's self-evaluation calibration—an important safety-relevant property for autonomous coding agents.

4 Why Hidden Functional Tests

The choice of evaluation methodology is not incidental—it determines what findings are possible. We contrast the hidden-test approach with alternatives:

vs. Code similarity (BLEU, CodeBLEU, edit distance). Refactoring *should* change code structure significantly. High similarity to the original may indicate the agent did nothing useful. Low similarity is expected and desirable, provided behavior is preserved. Metrics such as CodeBLEU [Ren et al., 2020] were designed for code generation, but for refactoring the desired output may be structurally different from the input while remaining behaviorally equivalent.

vs. LLM-as-judge. LLM-as-judge methods can approximate human preference judgments in open-ended settings [Zheng et al., 2023], but using another LLM to evaluate refactoring quality introduces model-specific biases and makes it difficult to distinguish genuine behavioral preservation from superficial plausibility. More fundamentally, an LLM judge must itself be evaluated: if we rely on a model's judgment of behavioral preservation, we need evidence that the judge is reliable on that exact task. This creates a recursive evaluation problem unless the judge is anchored to an external ground-truth signal. Hidden tests provide that signal directly. They also detect subtle regressions (e.g., a missing event handler, an off-by-one in a loop) with the precision of executable assertions.

vs. Static analysis only. Compilation success is necessary but not sufficient. A file can compile perfectly while rendering nothing, or while having broken event handlers. In our production system, compilation-based metrics consistently overstated quality.

vs. Agent self-reported success. The agent's own judgment of success is unreliable (Section 5). An agent that reports 90% success while achieving 60% behavioral preservation is more dangerous than one that honestly reports failure, because it undermines trust in the overall system.

The hidden-test approach is not novel in software engineering—Opdyke [1992], Griswold and Notkin [1993], and Fowler [1999] established behavioral

preservation as the correctness criterion for refactoring, and SafeRefactor-style systems use tests to detect behavioral changes [Soares et al., 2013]. RefactorBench-JS adapts this principle to the LLM agent setting, where the refactoring agent is a stochastic black box rather than a deterministic program transformation.

5 Baseline Results

To demonstrate RefactorBench-JS and establish initial baselines, we evaluate Anthropic Claude and Google Gemini models across two tool configurations.

5.1 Agent Configuration

The refactoring agent is a multi-turn completion agent with the following tools: `write_file` (create/update files), `remove_file` (delete files), `get_all_errors` (compilation error checking with pre-existing error filtering), and `finish_refactoring` (termination signal). The `run_unit_test` tool (execute a test file via Vitest or Jest) is toggled as the experimental variable.

All agents use temperature 0, `tool_choice=required`, a maximum of 50 tool calls per attempt, and up to 3 retry attempts with fresh conversation state. The system prompt includes the target file's content, platform conventions, import alias rules, and few-shot examples of correct refactoring.

The public repository contains the benchmark fixtures, hidden tests, paper source, and sanitized result data. It does not include our private production agent harness or proprietary system prompts; external users can evaluate RefactorBench-JS by applying their own refactoring harness to the committed fixtures and scoring against the hidden tests.

5.2 Experimental Design

We evaluate a 7×2 factorial design:

- **Models:** Anthropic Claude Sonnet 4.6, Anthropic Claude Opus 4.6, Anthropic Claude Opus 4.7, Google Gemini 2.0 Flash, Google Gemini 2.5 Flash, Google Gemini 2.5 Pro, Google Gemini 3.0 Pro
- **Tool:** Test runner enabled vs. disabled

Each condition is evaluated on the same 123 fixtures with hidden tests. Because every condition runs on the same fixtures, file complexity is controlled by design. Model names are reported using the provider/routing aliases recorded by the eval store at run time; the sanitized result files preserve those aliases along with source export identifiers and timestamps. Reproducing baseline rows requires matching the model snapshot, harness settings, tool set, retry policy, and scoring runtime.

5.3 Results

Table 3 reports the primary results with 95% Wilson score confidence intervals [Wilson, 1927] on the "Passes Tests" rate.

Table 3: Baseline results on the full RefactorBench-JS run ($n = 123$ fixtures per condition). "Passes Tests" is the number of fixtures whose hidden tests pass after refactoring. 95% Wilson score CIs are reported on the pass rate. Duration is wall-clock time per fixture in minutes.

Model	Test Runner	Pass / 123	Rate (95% CI)	Avg Min
Sonnet 4.6	Enabled	29	23.6% [16.9, 31.8]	8.9
Sonnet 4.6	Disabled	17	13.8% [8.8, 21.0]	6.6
Opus 4.6	Enabled	28	22.8% [16.2, 30.9]	12.1
Opus 4.6	Disabled	21	17.1% [11.4, 24.7]	6.7
Opus 4.7	Enabled	17	13.8% [8.8, 21.0]	7.6
Opus 4.7	Disabled	16	13.0% [8.2, 20.1]	6.5
Gemini 2.0 Flash	Enabled	13	10.6% [6.3, 17.2]	5.5
Gemini 2.0 Flash	Disabled	16	13.0% [8.2, 20.1]	4.6
Gemini 2.5 Flash	Enabled	18	14.6% [9.5, 21.9]	7.3
Gemini 2.5 Flash	Disabled	15	12.2% [7.5, 19.1]	4.4
Gemini 2.5 Pro	Enabled	21	17.1% [11.4, 24.7]	14.9
Gemini 2.5 Pro	Disabled	22	17.9% [12.1, 25.6]	18.9
Gemini 3.0 Pro	Enabled	31	25.2% [18.4, 33.5]	9.5
Gemini 3.0 Pro	Disabled	22	17.9% [12.1, 25.6]	13.8

Note on precision: All conditions use the same 123 fixtures. Wilson score intervals are reported to make marginal sampling uncertainty explicit; Table 4 additionally reports paired test-runner comparisons on the same fixture set. The Gemini 3.0 Pro enabled row includes one explicit terminal failure for a fixture that repeatedly terminated before exporting a result; it is counted as a failed refactoring. The Sonnet 4.6 disabled row completed with all 123 hidden-test scores exported, despite the cloud wrapper returning a late termination error after result export.

Table 4: Paired test-runner effect by model. "Enabled-only" counts fixtures that passed only when the test runner tool was available; "disabled-only" counts fixtures that passed only without it. p is a two-sided exact McNemar/binomial test over discordant fixture pairs.

Model	**Δ pp**	**Enabled-only**	**Disabled-only**	p
Sonnet 4.6	+9.8	13	1	0.0018
Opus 4.6	+5.7	8	1	0.0391
Opus 4.7	+0.8	1	0	1.0000
Gemini 2.0 Flash	-2.4	1	4	0.3750
Gemini 2.5 Flash	+2.4	4	1	0.3750
Gemini 2.5 Pro	-0.8	5	6	1.0000
Gemini 3.0 Pro	+7.3	9	0	0.0039

These paired tests are reported as model-level diagnostics rather than a multiplicity-adjusted family of confirmatory hypotheses.

5.4 Agent Calibration

Table 5 reports the divergence between agent-reported success (the agent signaled completion via `finish_refactoring` with `success=true`) and actual behavioral preservation (hidden tests pass). The *false confidence rate*—the fraction of agent-reported successes that fail hidden tests—measures how much the agent overestimates its own correctness.

High false confidence rates are practically important: an agent that reports success while breaking behavior is more dangerous than one that honestly reports failure, because downstream systems (and users) will trust the result. A small number of fixtures pass hidden tests even when the agent does not report success; therefore false confidence is computed from the intersection of reported-success and hidden-test outcomes, not by simply subtracting aggregate pass counts.

5.5 Operational Metadata

Table 6 reports operational metadata captured by the internal eval store. Costs are measured from the refactoring cost score emitted per fixture; durations are wall-clock times from fixture start to score export. The final column counts fixtures where the agent did not report successful completion, regardless of hidden-test outcome. Turn and tool-call telemetry is also present for the newer exports, but older Anthropic rows predate that telemetry patch,

Table 5: Agent calibration on the full run. "Reported" = agent signaled success. "Actual" = hidden tests pass. "False-confidence failures" are reported-success outcomes whose hidden tests fail; the rate divides that count by reported successes.

Model	Test Runner	Reported	Actual	False-conf. Fail.	False-conf. %
Sonnet 4.6	Enabled	62/123	29/123	35/123	56.5
Sonnet 4.6	Disabled	89/123	17/123	72/123	80.9
Opus 4.6	Enabled	53/123	28/123	26/123	49.1
Opus 4.6	Disabled	63/123	21/123	42/123	66.7
Opus 4.7	Enabled	62/123	17/123	46/123	74.2
Opus 4.7	Disabled	60/123	16/123	45/123	75.0
Gemini 2.0 Flash	Enabled	28/123	13/123	20/123	71.4
Gemini 2.0 Flash	Disabled	39/123	16/123	25/123	64.1
Gemini 2.5 Flash	Enabled	82/123	18/123	64/123	78.0
Gemini 2.5 Flash	Disabled	91/123	15/123	76/123	83.5
Gemini 2.5 Pro	Enabled	108/123	21/123	88/123	81.5
Gemini 2.5 Pro	Disabled	107/123	22/123	86/123	80.4
Gemini 3.0 Pro	Enabled	98/123	31/123	68/123	69.4
Gemini 3.0 Pro	Disabled	76/123	22/123	55/123	72.4

so we omit it from the aggregate table.

5.6 Gap to Perfect Behavioral Preservation

Table 7 decomposes each condition's gap to 100% hidden-test success. We distinguish reported non-success failures, where the agent did not signal successful completion and the hidden tests failed, from reported-success failures, where the agent signaled success but the hidden tests failed. The latter category is especially important: these are refactorings that would appear complete to an orchestration layer unless hidden tests are run.

The failure modes differ by model class. Smaller Gemini 2.0 Flash configurations are dominated by reported non-success failures, while stronger Gemini 2.5/3.0 configurations more often complete and report success despite failing hidden behavioral tests. This suggests that scaling model capability alone does not eliminate the need for external behavioral verification: the primary residual failure mode shifts from non-success to confident but behaviorally incorrect completion.

Table 6: Operational metadata for the full RefactorBench-JS run.

Model	Test Runner	Avg Min	Avg $/Fixture	Total $	Non-success
Sonnet 4.6	Enabled	8.9	1.373	168.8	61
Sonnet 4.6	Disabled	6.6	2.907	357.5	34
Opus 4.6	Enabled	12.1	1.921	236.3	70
Opus 4.6	Disabled	6.7	1.357	166.9	60
Opus 4.7	Enabled	7.6	2.078	255.6	61
Opus 4.7	Disabled	6.5	1.807	222.2	63
Gemini 2.0 Flash	Enabled	5.5	0.005	0.6	95
Gemini 2.0 Flash	Disabled	4.6	0.006	0.7	84
Gemini 2.5 Flash	Enabled	7.3	0.320	39.4	41
Gemini 2.5 Flash	Disabled	4.4	0.206	25.3	32
Gemini 2.5 Pro	Enabled	14.9	1.572	193.3	15
Gemini 2.5 Pro	Disabled	18.9	1.561	192.0	16
Gemini 3.0 Pro	Enabled	9.5	3.486	428.7	25
Gemini 3.0 Pro	Disabled	13.8	2.171	267.0	47

5.7 Error Taxonomy

To better understand the residual gap, we bucket failed fixture outcomes by the first recognizable failure mode in the structured scorer metadata. This taxonomy is heuristic: it is derived from agent reported-success status, non-triviality scores, compile/static scores, and regular-expression matching over hidden-test failure messages. It is nevertheless useful for identifying where engineering effort should go next.

The dominant failure mode is not subtle behavioral drift, but basic completion/reporting and syntactic correctness. However, the distribution is model-dependent. Opus and Gemini 2.0 Flash are reported-non-success-heavy, while Gemini 2.5 Pro and Gemini 3.0 Pro more often complete but fail through syntax/parse or module-boundary mistakes. This reinforces the benchmark's central claim: behavior-preserving refactoring requires both reliable completion and external behavioral verification.

Fixture-level hardness is highly skewed. Across the 123 fixtures, 86 passed in zero of the 14 conditions, while 9 passed in all 14. The zero-pass tier is not random: it is concentrated in large UI fixtures. Zero-pass targets average 1,294 physical lines (median 1,147), compared with 372 average lines (median 267) for fixtures that passed in at least one condition. No algorithm/data fixture was in the zero-pass tier; every pure algorithm/data target passed in at least one model/tool condition.

These failures suggest a qualitative boundary in current refactoring agents. On smaller pure-code fixtures, at least one model/tool setting usually finds a

Table 7: Failure-mode decomposition for the gap to 100% hidden-test success. "Gap" is 123 − Passes Tests. "Non-success failures" are failed hidden-test outcomes where the agent did not report successful completion; "reported-success failures" are failed hidden-test outcomes where the agent nevertheless reported success.

Model	Test Runner	Pass	Gap	Non-succ. Fail.	Reported-Succ. Fail.
Sonnet 4.6	Enabled	29/123	94	59	35
Sonnet 4.6	Disabled	17/123	106	34	72
Opus 4.6	Enabled	28/123	95	69	26
Opus 4.6	Disabled	21/123	102	60	42
Opus 4.7	Enabled	17/123	106	60	46
Opus 4.7	Disabled	16/123	107	62	45
Gemini 2.0 Flash	Enabled	13/123	110	90	20
Gemini 2.0 Flash	Disabled	16/123	107	82	25
Gemini 2.5 Flash	Enabled	18/123	105	41	64
Gemini 2.5 Flash	Disabled	15/123	108	32	76
Gemini 2.5 Pro	Enabled	21/123	102	14	88
Gemini 2.5 Pro	Disabled	22/123	101	15	86
Gemini 3.0 Pro	Enabled	31/123	92	24	68
Gemini 3.0 Pro	Disabled	22/123	101	46	55

behavior-preserving decomposition. On large UI and mobile fixtures, failures are systemic: 47.2% of zero-pass outcomes are reported non-success, while most of the remaining failures occur after the agent reports success but leaves syntax, parse, import/export, or UI behavior errors. This is evidence that the benchmark's hardest tier stresses multi-file boundary management and UI/runtime preservation, not just isolated algorithmic reasoning.

The public test contracts clarify what agents are breaking in the zero-pass tier. For example, the hardest `expense_tracker` fixture has 111 tests and 174 assertions covering sidebar navigation, transaction modals, filtering, sorting, localStorage persistence, export behavior, and computed financial summaries. `financial_portfolio` checks portfolio totals, sector and asset allocation, watchlists, alerts, filtering, sorting, and transaction forms. Mobile fixtures such as `grocery_mobile_app` and `sports_schedule` add authentication state, router behavior, mocked API calls, accessibility labels, and async loading states. Thus the zero-pass tier fails because agents do not preserve many interdependent UI, state, module, and side-effect behaviors after decomposition.

Table 8: Aggregate error buckets across all failed model/tool/fixture outcomes in the full run (1,436 total failed outcomes). Buckets are heuristic and mutually exclusive by priority order.

Error Bucket	**Count**	**Share of Failures**
Reported non-success	688	47.9%
Syntax or parse failure	422	29.4%
Module import/export failure	220	15.3%
DOM/query/UI mismatch	66	4.6%
Assertion-level behavior mismatch	16	1.1%
Runtime reference/type error	10	0.7%
Non-triviality failure	11	0.8%
Other hidden-test failure	3	0.2%

Table 9: Error buckets by model family, aggregated across test-runner enabled and disabled conditions. Counts are failed fixture outcomes; each model has up to 246 outcomes across the two tool settings. The "Other" column merges non-triviality failures and other hidden-test failures.

Model	**Fail**	**Non-succ.**	**Syntax**	**Import**	**DOM/UI**	**Assert.**	**Runtime**	**Other**
Sonnet 4.6	200	93	42	46	12	4	1	2
Opus 4.6	197	129	16	36	14	1	1	0
Opus 4.7	213	122	65	24	0	2	0	0
Gemini 2.0 Flash	217	172	8	20	9	1	4	3
Gemini 2.5 Flash	213	73	98	34	5	2	1	0
Gemini 2.5 Pro	203	29	104	38	17	4	2	9
Gemini 3.0 Pro	193	70	89	22	9	2	1	0

5.8 Analysis

Finding 1: Tool augmentation effects are model-dependent. The test runner tool has model-dependent effects on hidden-test pass rate. It improves Sonnet 4.6 by 9.8 percentage points, Opus 4.6 by 5.7 points, Gemini 3.0 Pro by 7.3 points, and Gemini 2.5 Flash by 2.4 points. It is nearly neutral for Opus 4.7 (+0.8 points), and directionally negative for Gemini 2.0 Flash (-2.4 points) and Gemini 2.5 Pro (-0.8 points). In paired exact McNemar tests over the shared fixture set (Table 4), the enabled-vs-disabled difference is strongest for Sonnet 4.6, Opus 4.6, and Gemini 3.0 Pro; the remaining model-level deltas should be interpreted descriptively.

This interaction effect—the same tool helping some models while not helping others—is a central empirical finding. It was only discoverable because RefactorBench-JS measures behavioral preservation directly. A

Table 10: Hardest fixtures by hidden-test failures across all 14 model/tool conditions. Rows are ranked by failure count, with average duration as a tie-breaker. Because 86 fixtures failed in all 14 conditions, this table should be read as representative of the hardest tier; the full ranking is released in `data/eval-results/refactorbench_js_fixture_hardness.csv`.

Fixture	Category	LOC	Passes	Avg. Min.	Dominant Failure Bucket
`expense_tracker`	Web UI	3,769	0/14	43.8	Reported non-success
`grocery_mobile_app`	Mobile	1,648	0/14	28.7	Reported non-success
`sports_schedule`	Mobile	1,287	0/14	24.2	Reported non-success
`wave_chat`	Web UI	1,953	0/14	23.0	Reported non-success
`learning_journey`	Mobile	1,007	0/14	21.8	Module import/export failure
`mobile_game`	Mobile	289	0/14	18.5	Reported non-success
`seo_platforms_data`	Utility/API	962	0/14	16.5	Assertion-level behavior mismatch
`financial_search`	Mobile	925	0/14	16.0	Reported non-success
`block_blast_game`	Mobile	509	0/14	15.8	Reported non-success
`financial_portfolio`	Web UI	998	0/14	15.6	Syntax or parse failure

Table 11: Composition of the zero-pass fixture tier. Failure counts are aggregated across all failed outcomes in the tier; each zero-pass fixture contributes 14 failed outcomes. Categories are heuristic labels inferred from target paths and file contents.

Category	Fixtures	Share	Median LOC	Largest Failure Buckets
React web UI	57	66.3%	1,204	Syntax/parse (366); reported non-success (333); import/export (78)
Mobile / React Native	26	30.2%	751	Reported non-success (221); import/export (58); syntax/parse (55)
Utility/API	3	3.5%	402	Import/export (15); reported non-success (14); assertion mismatch (11)

compilation-based metric would not have detected the difference, as many incorrect refactorings still compile.

Finding 2: Latency varies substantially by model and configuration. Average wall-clock duration ranges from 4.4 minutes per fixture for Gemini 2.5 Flash without the test runner to 18.9 minutes for Gemini 2.5 Pro without the test runner. Anthropic rows range from 6.5 to 12.1 minutes per fixture; Gemini rows range from 4.4 to 18.9 minutes. For interactive applications where users wait for refactoring, this operational variation matters alongside hidden-test pass rate.

Hypothesis: Iterative tool calling outperforms batch tool calling. We informally observed that Gemini 2.5 Pro tends to accumulate reasoning and emit batches of tool calls, while Sonnet models call tools iteratively—one call, observe the result, decide the next action. We also observed that Sonnet

Table 12: Behavioral contract families exercised by the released tests for the 86 zero-pass fixtures. Labels are heuristic and derived from public test source text; a fixture can exercise multiple contract families.

Contract Family	**Zero-Pass Fixtures**
Computed domain state	84/86
Initial render and visible copy	82/86
Search, filtering, and sorting	80/86
Module exports and data shape	79/86
Forms, modals, and CRUD flows	77/86
Navigation and view switching	77/86
API, auth, and external side effects	76/86
Accessibility and semantic selectors	62/86
Browser or device persistence	53/86
Async behavior and timers	48/86
Mobile/native runtime behavior	25/86

models are more likely to invoke the build checker tool (`get_all_errors`) proactively before signaling completion, suggesting better self-verification behavior. We hypothesize that the iterative pattern is more suitable for refactoring, where each file write changes the filesystem state that subsequent decisions depend on. These observations are qualitative and unquantified; confirming them requires measuring tool-call-per-turn distributions and tool-type usage frequencies, which we plan as future work.

5.9 Retry-Adjusted Cost Analysis

Table 13 reports expected cost and time per successful refactoring under a simple retry model with up to 3 independent attempts, stopping early on success. Let p denote the per-attempt hidden-test pass rate and c the observed average cost per fixture. The expected number of billed attempts is:

$$E[\text{attempts}] = 1 \cdot p + 2 \cdot (1-p) \cdot p + 3 \cdot (1-p)^2$$

The final term covers all cases where the first two attempts fail: the third attempt is run and billed whether it succeeds or fails. The probability of eventual success within 3 attempts is $P_{\text{succ}} = 1 - (1-p)^3$, and the expected cost per successful refactoring is $E[\text{attempts}] \cdot c \;/\; P_{\text{succ}}$.

This model assumes independent, identically distributed trial outcomes. In practice, retries use fresh conversation state but operate on the same file, so failures may be correlated with file-specific difficulty. The i.i.d. assumption

likely understates expected cost for files that are inherently difficult to refactor.

Table 13: Expected cost and time per successful refactoring, accounting for up to 3 retries. $P_{\text{succ}} = 1 - (1 - p)^3$ is the probability of at least one success within 3 i.i.d. attempts.

Model	**Test Runner**	P_{succ} **(%)**	**E[$/succ.]**	**E[min/succ.]**
Sonnet 4.6	Enabled	55.4	5.82	37.7
Sonnet 4.6	Disabled	36.0	21.03	47.8
Opus 4.6	Enabled	53.9	8.44	53.2
Opus 4.6	Disabled	43.0	7.95	39.2
Opus 4.7	Enabled	36.0	15.03	55.0
Opus 4.7	Disabled	34.2	13.89	50.0
Gemini 2.0 Flash	Enabled	28.5	0.05	52.0
Gemini 2.0 Flash	Disabled	34.2	0.05	35.4
Gemini 2.5 Flash	Enabled	37.8	2.19	49.9
Gemini 2.5 Flash	Disabled	32.3	1.69	36.1
Gemini 2.5 Pro	Enabled	43.0	9.21	87.3
Gemini 2.5 Pro	Disabled	44.6	8.73	105.7
Gemini 3.0 Pro	Enabled	58.2	13.83	37.7
Gemini 3.0 Pro	Disabled	44.6	12.14	77.2

Retry-adjusted cost changes the ranking relative to raw pass rate. Gemini 2.0 Flash remains cheapest because its per-fixture cost is extremely low, but it also has low probability of success within three attempts. Gemini 3.0 Pro with the test runner has the highest three-attempt success probability in this run, but also the highest observed cost per fixture. These numbers should be read as an operational planning model rather than a claim about independent repeated trials on identical files.

6 Production Impact

RefactorBench-JS was developed to guide decisions for a production refactoring system. This section gives production context for the benchmark-informed decisions. The results are observational: changes were deployed incrementally without controlled A/B testing, and should be read as operational context rather than experimental claims.

6.1 Tool Evolution

The refactoring subagent's tool set evolved through four versions, each motivated by observed failure modes:

- **V0 (write-only):** 35–40% success rate. Failures included premature stopping, infinite looping (re-writing files already in context), and never modifying the original file.
- **V1 (+termination tool):** Addressed premature stopping by providing an explicit completion action.
- **V2 (+build checker):** Error diffing (showing only *new* errors, not pre-existing ones) was critical; without it, the agent would chase unrelated compilation errors.
- **V3 (+test runner):** Enabled the agent to self-verify via its own tests. As RefactorBench-JS showed, this tool's effect is highly model-dependent.

6.2 Model Selection

Guided by RefactorBench-JS results, we migrated from Gemini 2.5 Pro to Anthropic Sonnet models with the test runner tool enabled. Combined with the tool evolution and additional engineering work (tool-call deduplication to prevent loops, lazy file synchronization to reduce latency, prompt improvements including few-shot examples and a 6-step procedural scaffold), observed production success rates improved from ∼65% to above 97%. This was an observational production change, not a controlled A/B test; the benchmark helped identify promising directions, but the production improvement should not be read as a causal estimate for any single intervention.

7 Related Work

LLM refactoring. Prior work on LLM-assisted refactoring focuses on single-turn transformations: extract method, renaming, or type migration [Shirafuji et al., 2023, Pomian et al., 2024]. RefactorBench-JS evaluates *multi-file, multi-turn, agentic* refactoring—a qualitatively different task requiring sequential decision-making.

Agentic coding benchmarks. SWE-bench [Jimenez et al., 2024] evaluates agents on GitHub issues. SWE-agent [Yang et al., 2024] showed that tool design significantly affects agent performance. OpenHands [Wang et al., 2024] provides an open platform for development agents. Gautam et al.'s RefactorBench [Gautam et al., 2025] evaluates stateful reasoning on 100 handcrafted multi-file refactoring tasks in open-source repositories; RefactorBench-JS is distinct in scope, focusing on JavaScript/React decomposition fixtures, hidden behavioral tests, calibration, operational metadata, and model×tool interactions. Aider's refactoring benchmark [Gauthier, 2024a] uses a test-based methodology similar to ours, applied to Python; their polyglot benchmark [Gauthier, 2024b] extends code-editing evaluation across languages.

Multi-turn reliability. Liu et al. [2024] show that LLMs struggle with information placed in the middle of long contexts, a related but not identical pressure point for long refactoring traces. Chen et al. [2024] benchmark tool-calling reliability across models, finding variation consistent with our qualitative observations.

Tool augmentation. Toolformer [Schick et al., 2023] and Qin et al. [2023] establish the benefits of tool use. TPTU [Ruan et al., 2024] identifies tool selection and invocation as distinct failure modes. Our contribution is evidence that tool benefit is model-dependent—some models degrade with additional tools.

Refactoring correctness. Opdyke [1992], Griswold and Notkin [1993], and Fowler [1999] established behavioral preservation as the correctness criterion. SafeRefactor-style systems use test generation and differential execution to detect behavioral changes [Soares et al., 2013]. RefactorBench-JS adapts this principle to the LLM agent setting with hidden holdout tests.

8 Discussion

8.1 Implications for Eval Design

RefactorBench-JS's most transferable contribution is methodological. We advocate three principles for evaluating behavior-preserving code transformations:

1. **Evaluate functional properties, not structural ones.** Compilation is necessary but not sufficient. Code similarity is actively misleading for refactoring.

2. **Use hidden tests that exercise real behavior.** Tests should be invisible to the agent and should assert observable behavior rather than implementation structure.

3. **Measure agent calibration.** Track divergence between agent-reported success and actual behavioral preservation.

8.2 The Model×Tool Interaction

The finding that a test runner tool can help some models more than others has practical implications: model and tool choices should be evaluated jointly, not independently. We hypothesize that the interaction reflects differences in iterative reasoning ability, goal maintenance under tool abundance, and appropriate use of termination signals, but confirming these mechanisms requires future work.

8.3 Limitations and Future Work

- **Corpus scope and sourcing.** The corpus is JavaScript/React-focused and reflects realistic refactoring examples from a single production context. This provides practical complexity, but may introduce domain bias. Generalization to other languages, frameworks, and refactoring types is unknown and planned.

- **Public holdout contamination.** Hidden tests are hidden from the agent during an evaluation attempt, but they are public in the released repository for transparency and auditability. Future model snapshots may train on the released fixtures or tests; long-lived benchmark use should therefore rely on versioned releases, contamination checks, or private refresh splits.

- **No tool ablation.** We evaluate only two tool configurations (all tools ± test runner). A full ablation crossing all tool subsets with all models would be more informative.

- **Qualitative behavioral claims.** Tool-calling pattern differences are observed but not quantified. Future work should report tool-calls-per-turn distributions.

- **Test suite authorship.** Hidden test suites vary in authorship and coverage. Broader human review, external contributions, and independently authored tests would reduce potential bias.

We invite the community to evaluate their own models and agent designs against RefactorBench-JS and contribute additional files and test suites. The benchmark and sanitized per-fixture result data are available at https://github.com/Create-Inc/refactor-bench.

9 Conclusion

We introduced RefactorBench-JS, an open benchmark of 123 scored fixtures for evaluating LLM agents on behavior-preserving JavaScript/React code decomposition. The benchmark is grounded in a simple principle—behavioral preservation is a functional property best verified by functional tests—and implements this through hidden unit test suites that are never shown to the refactoring agent.

Baseline evaluations reveal that tool augmentation effects, calibration, cost, latency, and failure modes vary substantially across model families. These findings, which are only visible with behavioral scoring and structured metadata, demonstrate why proper eval design is a prerequisite for meaningful agent development.

RefactorBench-JS is publicly available in the public repository, including sanitized per-fixture result data for the full run reported here.

References

Z. Chen, S. Du, B. Zhao, Y. Shi, J. Geng, Y. Xu, J. Fu, and S. Li. Benchmarking and improving tool-calling reliability of large language models. *arXiv preprint arXiv:2406.05015*, 2024.

M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

P. Gauthier. Aider refactoring benchmark. https://aider.chat/docs/leaderboards/refactor.html, 2024. Accessed 2026-05-20.

P. Gauthier. Aider polyglot benchmark. https://aider.chat/2024/12/21/polyglot.html, 2024. Accessed 2026-05-20.

D. Gautam, S. Garg, J. Jang, N. Sundaresan, and R. Zilouchian Moghaddam. RefactorBench: Evaluating stateful reasoning in language agents through code. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2025.

W. G. Griswold and D. Notkin. Automated assistance for program restructuring. *ACM Transactions on Software Engineering and Methodology*, 2(3):228–269, 1993.

C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2024.

N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024.

W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

M. Pomian, P. Turbak, R. Ramasubbu, N. Tsantalis, and D. Dig. Together we go further: LLMs and IDE static analysis for extract method refactoring. *arXiv preprint arXiv:2401.15298*, 2024.

Y. Qin, S. Liang, Y. Ye, K. Zhu, L. Yan, Y. Lu, Y. Lin, X. Cong, X. Tang, B. Qian, S. Zhao, R. Tian, R. Xie, J. Zhou, M. Gerstein, D. Li, Z. Liu, and M. Sun. Tool learning with foundation models. *arXiv preprint arXiv:2304.08354*, 2023.

S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma. CodeBLEU: A method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020.

J. Ruan, Y. Chen, B. Zhang, Z. Xu, T. Bao, G. Du, S. Shi, H. Mao, X. Zeng, and R. Zhao. TPTU: Large language model-based AI agents for task planning and tool usage. *arXiv preprint arXiv:2308.03427*, 2024.

T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, L. Zettlemoyer, N. Cancedda, and T. Scialom. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

A. Shirafuji, Y. Watanobe, T. Ito, M. Morishita, and Y. Nakamura. Refactoring programs using large language models with few-shot examples. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2023.

G. Soares, R. Gheyi, D. Serey, and T. Massoni. Automated behavioral testing of refactoring engines. *IEEE Transactions on Software Engineering*, 39(2):147–162, 2013.

X. Wang, B. Li, Y. Song, F. F. Xu, X. Tang, M. Zhuge, J. Pan, Y. Song, B. Li, J. Singh, H. H. Tran, F. Li, R. Ma, M. Zhang, B. Yu, J. Bisk, C. Xiong, T. Yu, G. Neubig, and Y. Li. OpenHands: An open platform for AI software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024.

E. B. Wilson. Probable inference, the law of succession, and statistical inference. *Journal of the American Statistical Association*, 22(158):209–212, 1927. doi:10.1080/01621459.1927.10502953.

J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press. SWE-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*, 2024.

L. Zheng, W.-L. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. Xing, H. Zhang, J. E. Gonzalez, and I. Stoica. Judging LLM-as-a-judge with MT-Bench and Chatbot Arena. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.