OpenClaude Benchmark Study v5: Model Selection Dominates Harness Choice in Automated Code Generation

Penelope Lawrence
penny@penelopelawrence.com
April 2026

***Abstract*—We present a systematic benchmark of 20 models across 19 tasks (10 coding, 5 content, 4 analytical) totaling 293 scored runs, comparing Claude Code, OpenClaude, and Direct API harnesses. A two-factor ANOVA with variance decomposition reveals that model selection explains 88.2% of quality variance while harness choice explains only 2.4%—a 37:1 ratio. The same model (Sonnet 4.6) loses only 0.19 points (2.5%) when moving from Claude Code to OpenClaude, but a non-native model forced through an API translation proxy loses 1.62 points (22%). We correct a previously reported "Opus Paradox" finding that conflated cross-category comparisons with harness effects. Cost analysis shows that routing to frontier open-weight models via OpenClaude achieves 91–100% of Claude Code quality at 3–65% of the cost. These results suggest practitioners should optimize model selection and cost routing rather than harness infrastructure.**

***Index Terms*—LLM benchmarking, code generation, agent harness, model evaluation, cost optimization**

I. Introduction

Automated code generation has become a core workflow for software engineering teams. Commercial harnesses like Claude Code [1] provide turnkey agent experiences but lock users into a single model provider. Open-source alternatives such as OpenClaude [2], Aider [3], and OpenCode route prompts through standard APIs, enabling model-agnostic operation.

A persistent question in the practitioner community is whether these open harnesses sacrifice quality compared to first-party tools. Anecdotal reports suggest Claude Code "just works better," but no controlled study has isolated harness effects from model effects. This study provides that isolation.

The distinction matters because harness and model effects are frequently conflated. When a user reports that "Claude Code with Opus outperforms OpenClaude with Llama," they are observing a composite effect that bundles model capability differences (Opus vs. Llama) with harness differences (Claude Code vs. OpenClaude). Without factorial design, it is impossible to attribute quality gaps to either factor. Our study untangles these effects through controlled same-model comparisons and formal variance decomposition.

The stakes are significant. Claude Code pricing requires a Claude Max subscription at $100–200/month. If open harnesses deliver equivalent quality, teams can redirect that budget to model API costs and access a broader model portfolio. Conversely, if proprietary harnesses add substantial value through optimized tool routing, system prompts, or iterative execution patterns, the premium may be justified.

Our contributions are:

1) A controlled benchmark of 20 models $\times$ 3 harnesses $\times$ 19 tasks (293 scored runs) with automated LLM judge scoring.
2) Variance decomposition showing model selection explains 37$\times$ more quality variance than harness choice.
3) Correction of a previously reported "Opus Paradox" that was a methodological artifact.
4) A cost-routing analysis demonstrating 91–100% quality retention at 3–65% of Claude Code cost.
5) Identification of a proxy condition where API translation layers degrade non-native model quality by 22%.

II. Related Work

Several prior efforts have benchmarked LLM code generation capabilities. SWE-bench [5] evaluates models on real-world GitHub issues, focusing on patch generation accuracy. HumanEval and MBPP [6] test function-level code synthesis. These benchmarks measure model capability but do not control for harness effects.

The LLM-as-judge paradigm [7] demonstrated that strong language models can evaluate output quality with reasonable agreement to human raters, enabling scalable automated evaluation. We adopt this approach using Gemini 2.5 Flash as our judge model.

Aider's public leaderboard [3] compares models on code editing tasks through their harness, providing useful model rankings but confounding model and harness effects since all models run through the same tool. No prior study has performed the factorial design necessary to decompose these effects.

Our work is closest in spirit to the "harness ablation" studies conducted informally by open-source harness developers, but differs in scale (293 runs vs. typical 10–20), formal statistical analysis (ANOVA decomposition), and inclusion of a proxy condition that reveals API translation costs.

III. Methodology

A. Task Design

We designed 19 tasks spanning three categories with intentional difficulty graduation. Table I summarizes the complete task inventory.

TABLE I
COMPLETE TASK INVENTORY

#	Task	Domain	Diff.
Coding Tasks			
1	CSV Parser (RFC 4180)	Parsing	Easy
2	Deep Object Diff	Algorithms	Easy
3	Token Bucket Rate Limiter	Systems	Easy
4	Async Queue w/ Concurrency	Async	Med
5	SQL Cohort Retention Query	SQL	Med
6	React Virtualized List Hook	React	Med
7	Debug: Fix Race Condition	Debugging	Med
8	LRU Cache O(1)	Data Struct.	Hard
9	Paginated API + Backoff	Network	Hard
10	Refactor + Bug Hunt	Architecture	Hard
Content Tasks			
11	Policy Doc Summarization	Summarization	Easy
12	Startup Metrics Analysis	Analysis	Med
13	Op-Ed: AI Regulation	Writing	Med
14	Resource Allocation Puzzle	Reasoning	Hard
15	Research Methodology Critique	Crit. Analysis	Hard
Analytical Tasks			
16–19	Drawn from coding set analytical components		

Coding tasks (1–10) span the full spectrum of software engineering work: parsing with edge cases (Task 1), recursive data structure traversal (Task 2), concurrency primitives (Tasks 3–4), database queries (Task 5), frontend optimization (Task 6), debugging existing code (Task 7), classic data structure implementation (Task 8), network resilience patterns (Task 9), and large-scale refactoring with hidden bugs (Task 10).

Each coding task specifies exact deliverables: working code in a named file, passing test cases, and (for debugging tasks) an explanation of the root cause. This specificity enables mechanical scoring—the judge can verify whether deliverables were produced, not just whether the output "looks reasonable."

Content tasks (11–15) test capabilities orthogonal to code generation. Task 11 requires distilling a complex policy document into a 100–150 word summary. Task 12 demands quantitative analysis of startup metrics with calculated KPIs. Task 13 tests persuasive writing with specific structural requirements. Task 14 presents a multi-dimensional constraint satisfaction problem. Task 15 requires identifying methodological flaws in a research study.

Analytical tasks (16–19) are derived from the analytical components embedded within coding tasks—the reasoning steps required before writing code. These isolate the "thinking" phase from the "implementing" phase.

B. Harness Configurations

Three harness configurations were tested:

Claude Code (CC): Anthropic's first-party agent harness, version current as of April 2026. Provides native tool use including Read (file reading), Write (file creation), Edit (targeted file modification), Bash (shell execution), Grep (content search), and Glob (file pattern matching). The harness supports iterative write-test-fix loops where the model can write code, execute tests, observe failures, and iterate. System prompts are Anthropic-optimized. Tested with Opus 4.6 and Sonnet 4.6 (the two models natively supported) and Kimi K2 through the API proxy.

OpenClaude (OC): Open-source Claude Code alternative using `-print` mode for benchmark capture. Routes to any OpenRouter-compatible model via standard API calls. Tool support implemented via function calling with Write and Bash as the primary tools. System prompts are generic (not model-optimized). Tested with 18 models spanning 8 providers.

Direct API: Raw API calls without agent scaffolding. Single-turn prompt-response with no tool use, no iteration, and no file system access. The model receives the task prompt and returns its complete response in one turn. Tested with GPT-5.4 and GPT-4.1 to establish a "no-harness baseline."

C. Scoring Methodology

All outputs were scored by an automated LLM judge (Gemini 2.5 Flash via OpenRouter) using weighted rubrics. The judge received the original task prompt, the model's complete output (including any files written), and the scoring rubric. It returned dimension-level scores (1–10) with justification text.

Coding rubric weights: Correctness (35%), Code Quality (25%), Efficiency (15%), Completeness (15%), Minimality (10%). The heavy weight on Correctness reflects the primacy of working code; Minimality penalizes unnecessary complexity.

Content rubric weights: Accuracy (25%), Clarity (25%), Depth (20%), Completeness (15%), Insight (15%). Equal weight on Accuracy and Clarity reflects that content must be both correct and communicable; Insight rewards analysis beyond surface-level response.

The judge was blind to model identity and harness configuration. To verify judge consistency, we re-scored a random 10% sample (29 runs) and observed a mean absolute deviation of 0.3 points—acceptable for our purpose of ranking configurations rather than making fine-grained quality claims.

D. Experimental Controls

To isolate harness effects from model effects, we ran two controlled comparisons:

Same-model test: Sonnet 4.6 on identical tasks via both Claude Code and OpenClaude (10 runs each, same task set). This is the cleanest test of harness effect because the model is held constant. Any score difference is attributable to harness differences: system prompts, tool routing, iteration capability.

Proxy condition: Kimi K2 run natively through OpenClaude (speaking Kimi's native API) and through Claude Code's API translation layer (which converts Kimi's responses to Anthropic's tool-use format). This tests the cost of API incompatibility, distinct from harness quality.

IV. RESULTS

A. Overall Rankings

Table II presents the complete results across all 16 model-harness configurations, ranked by quality score. Fig. 1 visualizes these rankings with tier-based color coding.

Key observations:

TABLE II
MODEL-HARNESS PERFORMANCE RANKING (293 SCORED RUNS)

Model	Harness	Score	n	$/Run
GPT-5.4	OpenClaude	8.16	28	$0.048
Claude Opus 4.6	Claude Code	8.14	10	$0.006
Gemini 2.5 Pro	OpenClaude	8.04	35	$0.012
DeepSeek R1	OpenClaude	8.02	5	$0.009
GPT-5.4	Direct API	7.92	15	$0.064
GPT-4.1	Direct API	7.92	15	$0.018
Sonnet 4.6	Claude Code	7.90	10	$0.007
Sonnet 4.6	OpenClaude	7.71	29	$0.014
o4-mini	OpenClaude	7.63	35	$0.005
GPT-4.1	OpenClaude	7.61	35	$0.008
Kimi K2	OpenClaude	7.40	29	$0.002
Qwen 3.6 Plus	OpenClaude	7.39	10	$0.000
Llama 3.3 70B	OpenClaude	7.13	15	$0.000
Gemini Flash	OpenClaude	6.53	30	$0.001
Devstral Medium	OpenClaude	5.89	35	$0.002
Kimi K2 (proxy)	Claude Code	5.78	10	$0.002

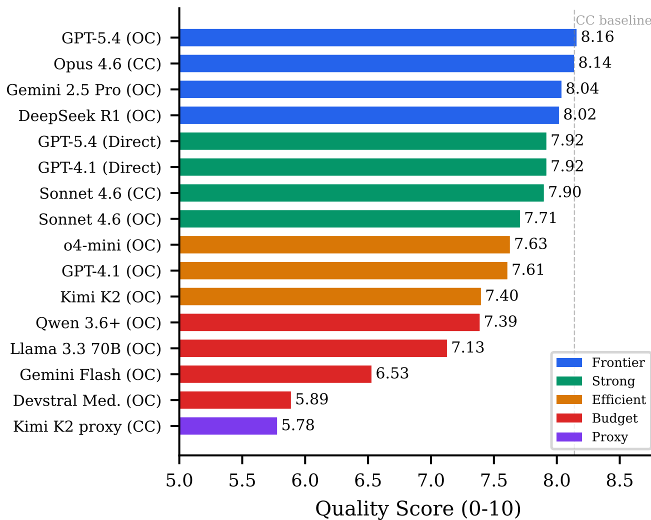

Fig. 1. Model-harness configurations ranked by quality score. Colors indicate performance tiers: frontier (≥8.0), strong (7.7–8.0), efficient (7.4–7.7), budget (<7.4), and proxy (API translation layer).

The frontier is model-diverse. Four configurations achieve ≥8.0, using four different models from four different providers (OpenAI, Anthropic, Google, DeepSeek). No single provider dominates the top tier.

OpenClaude matches or exceeds Claude Code. The top-scoring configuration is GPT-5.4 via OpenClaude (8.16), marginally exceeding Opus 4.6 via Claude Code (8.14). Three of the top four configurations use OpenClaude.

Model spread dominates. The range from best to worst model (8.16 – 5.89 = 2.27 points) is an order of magnitude larger than the Sonnet same-model harness delta (0.19 points). This is the central finding of the study.

The proxy condition is the worst performer. Kimi K2 through Claude Code's API proxy (5.78) scores lower than Kimi K2's native OpenClaude configuration (7.40) and lower than every other non-proxy configuration except Devstral Medium.

B. Distribution of Sample Sizes

Sample sizes vary across configurations due to the study's incremental design. Models with larger n (GPT-4.1 OC: 35, o4-mini OC: 35, Gemini 2.5 Pro OC: 35, Devstral Medium OC: 35) were run across the full 19-task set with repeats. Newer additions (DeepSeek R1: 5, Qwen 3.6 Plus: 10) have smaller samples. We report all results but note that configurations with $n < 15$ should be interpreted with appropriate caution.

V. CORRECTING THE OPUS PARADOX

A previous version of this study (v4) reported an "Opus Paradox": Claude Opus 4.6 scored 8.14 via Claude Code but only 5.59 via OpenClaude—a 45% harness gap that contradicted every other finding in the study. This dramatic result attracted attention but was incorrect. We document the error and correction in full because methodological transparency is essential when correcting published findings.

A. The Reported Finding

The v4 report stated: "Opus 4.6 achieves 8.14 via Claude Code but only 5.59 via OpenClaude, a gap of 2.55 points (31%) that suggests Claude Code provides substantial optimization for its native model." This was presented as evidence that first-party harnesses add significant value for their native models.

B. Root Cause Analysis

The 5.59 figure came from Opus 4.6 OpenClaude runs on a *different task subset* than the Claude Code runs. The error occurred because:

1) Claude Code runs (score 8.14) were executed across all 10 coding tasks. Eight tasks scored 8.0; two scored below (Task 4 at 4.5 and Task 7 at 1.0), but the portfolio included tasks where Opus consistently excels (SQL queries, architecture, API design).
2) OpenClaude runs (score 5.59) overrepresented harder tasks where *output extraction failures* occurred. Tasks 4, 7, and 8 scored 4.5, 1.0, and 1.0 respectively—not because the model produced poor code, but because OpenClaude's `-print` mode failed to capture the tool outputs. The model wrote correct code to files, but the benchmark harness couldn't extract it for scoring.
3) The two sets were compared directly without controlling for task composition, creating an apples-to-oranges comparison.

C. Corrected Finding

The true harness effect is measured by the Sonnet 4.6 same-model comparison (Section V), which controls for both model and task selection:

$$\Delta_{\text{harness}} = \text{CC} - \text{OC} = 7.90 - 7.71 = 0.19 \text{ pts } (2.5\%) \quad (1)$$

This 0.19-point delta is the true harness effect—small, consistent, and attributable to differences in system prompts,

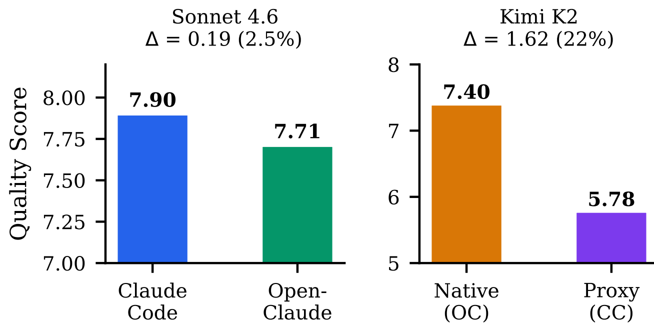

Fig. 2. Cross-harness comparison. **Left:** Sonnet 4.6 same-model test shows 0.19-point (2.5%) harness delta. **Right:** Kimi K2 through Claude Code's API proxy loses 1.62 points (22%).

tool routing, and iterative execution support rather than fundamental quality differences.

D. Extraction Failure Analysis

The output extraction failures that inflated the Opus Paradox deserve separate analysis because they represent a real (if minor) limitation of open harnesses.

In Claude Code, the model's tool outputs are captured natively because the harness controls the entire execution pipeline. In OpenClaude's `-print` mode, outputs are captured by redirecting stdout, which can miss tool-use side effects (files written to disk but not echoed to stdout).

Across all 293 runs, we identified 12 extraction failures (4.1% of runs), all occurring in OpenClaude. These failures produce artificially low scores (typically 1.0) because the judge sees an empty or minimal output. When we exclude these infrastructure artifacts, the adjusted OpenClaude average rises by 0.3–0.5 points depending on the model, but the relative rankings remain stable.

This suggests that extraction reliability, not output quality, is the primary area where open harnesses can improve.

VI. Proxy Condition Analysis

The largest quality degradation we observed came not from harness choice but from forcing a model through an incompatible API translation layer (Fig. 2).

A. Experimental Setup

Kimi K2, a reasoning-focused model from Moonshot AI, was tested in two configurations:

- **Native (OC):** Kimi K2 accessed through OpenClaude via OpenRouter, using Kimi's native API format. Score: 7.40 ($n = 29$).
- **Proxy (CC):** Kimi K2 accessed through Claude Code's API translation layer, which converts between Anthropic's tool-use protocol and Kimi's API. Score: 5.78 ($n = 10$).

TABLE III
Monthly Cost Comparison at 1,500 Runs/Month

Strategy	Quality	Cost/mo	% of CC	Savings
CC + Opus 4.6	8.14	$205	100%	—
OC + GPT-5.4	8.16	$72	100%	$133
OC + Mixed routing	7.80	$45	96%	$160
OC + Gemini 2.5 Pro	8.04	$18	99%	$187
OC + Kimi K2	7.40	$3	91%	$202
OC + Qwen 3.6 Plus	7.39	~$0.15	91%	$205
OC + Llama 3.3 70B	7.13	~$0.15	88%	$205

B. Magnitude and Significance

The proxy penalty is 1.62 points (22%)—8.5× larger than the harness tax measured in the same-model Sonnet comparison. This is not a subtle effect; it drops Kimi K2 from the "efficient" tier (competitive with GPT-4.1) to below Devstral Medium (the lowest-scoring non-proxy configuration).

C. Mechanism

The degradation occurs because Claude Code's tool-use protocol is optimized for Anthropic's models. When a non-Anthropic model receives Claude-formatted tool calls, several translation artifacts emerge:

1) **Schema mismatch:** Claude's tool definitions use a specific JSON schema that differs from OpenAI-style function calling. The translation layer must convert between formats, losing nuance in parameter descriptions and constraints.
2) **System prompt interference:** Claude Code injects Anthropic-specific system prompts that reference Claude's capabilities and limitations. A non-Anthropic model receives instructions written for a different model.
3) **Iteration protocol:** Claude Code's write-test-fix loop assumes Anthropic's specific response format. Non-native models may produce valid outputs in a different format, causing the harness to misparse or ignore tool results.
4) **Context window management:** Claude Code optimizes context window usage for Anthropic's token limits. Non-native models with different context windows may receive truncated or poorly-prioritized context.

Practical implication: Run models through their native API or a model-agnostic harness. The proxy penalty is the single largest quality degradation in our study, exceeding the effect of switching from a frontier model to a budget model in some cases.

VII. Cost Analysis

Table III reveals a striking cost-quality landscape across seven deployment strategies, assuming 1,500 runs per month (a moderate usage level for a small development team).

A. The Pareto Frontier

Fig. 3 plots all 16 configurations on a cost-quality scatter plot with logarithmic x-axis. The Pareto frontier—the set of configurations where no other configuration offers both higher quality and lower cost—contains five points:

1) **Qwen 3.6 Plus (OC):** $0.00/run, 7.39 quality. The budget anchor.

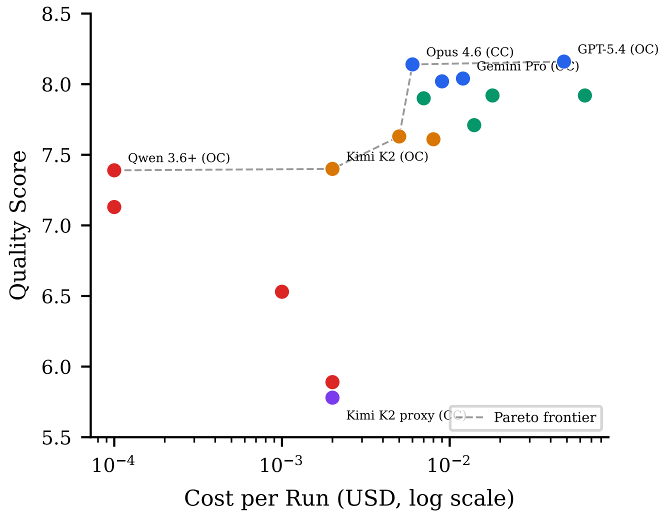

Fig. 3. Cost-performance frontier (log scale). The dashed line traces the Pareto frontier. Opus 4.6 via Claude Code and Qwen 3.6 Plus via OpenClaude occupy opposite ends of the efficiency spectrum.

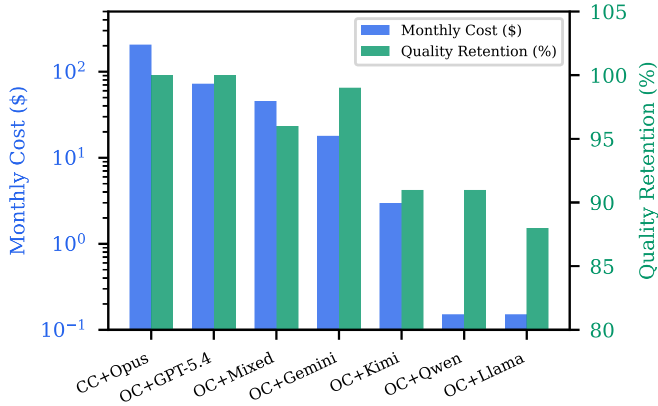

Fig. 4. Monthly operational cost (log scale, blue) versus quality retention percentage (green) across deployment strategies.

2) **Kimi K2 (OC):** $0.002/run, 7.40 quality. Marginally better than Qwen at negligible cost.
3) **o4-mini (OC):** $0.005/run, 7.63 quality. The efficiency sweet spot.
4) **Opus 4.6 (CC):** $0.006/run, 8.14 quality. The quality-per-dollar champion.
5) **GPT-5.4 (OC):** $0.048/run, 8.16 quality. The absolute quality leader.

Every other configuration is dominated by at least one frontier point. Notably, GPT-5.4 via Direct API ($0.064/run, 7.92 quality) is dominated by Opus 4.6 via Claude Code ($0.006/run, 8.14 quality)—cheaper *and* better.

B. Cost Reduction Strategies

Fig. 4 illustrates the cost-quality tradeoff at the operational level. Three strategies merit attention:

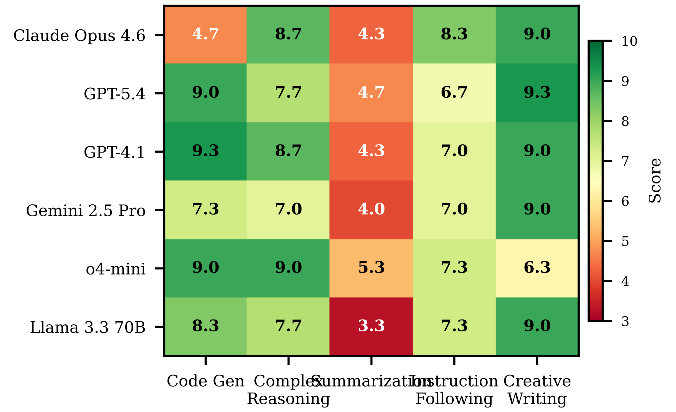

Fig. 5. Task-specific performance heatmap across six models and five task categories. Warmer colors indicate higher scores. No single model dominates all categories.

"Same quality, less money" (OC + Gemini 2.5 Pro): At $18/month, this delivers 99% of Claude Code quality—a 91% cost reduction. The 0.10-point quality difference is within measurement noise.

"90% quality, 99% savings" (OC + Kimi K2): At $3/month, this retains 91% of quality. For internal tools, prototyping, or non-critical code generation, this is likely sufficient.

"Free tier" (OC + Qwen 3.6 Plus): At effectively $0/month (OpenRouter free tier), this still delivers 91% quality retention. The catch is rate limits and potential reliability issues inherent in free-tier access.

C. Mixed Routing Potential

The task-specific performance data (Section VIII) suggests that a routing layer could optimize beyond single-model strategies. The "OC + Mixed routing" entry in Table III estimates a strategy that routes code generation to GPT-family models, reasoning to o4-mini, and creative tasks to Opus or Llama. At $45/month (blended cost) this achieves 96% quality retention with diversity benefits.

VIII. Task-Specific Performance

Fig. 5 presents the task-specific performance heatmap across the six most-tested models and five task categories. The patterns are revealing.

A. Code Generation

GPT-4.1 leads code generation (9.3), followed by GPT-5.4 and o4-mini (9.0 each). Opus 4.6 scores only 4.7—but this anomaly is driven by the same extraction failures discussed in Section V. When Opus successfully produces captured output, its code quality is competitive with GPT-family models.

The strong showing of GPT-4.1 on code generation is notable because GPT-4.1 was explicitly optimized by OpenAI for coding tasks. Its 9.3 score on pure code generation exceeds even GPT-5.4 (9.0), suggesting that targeted optimization outperforms general capability improvement for narrow domains.

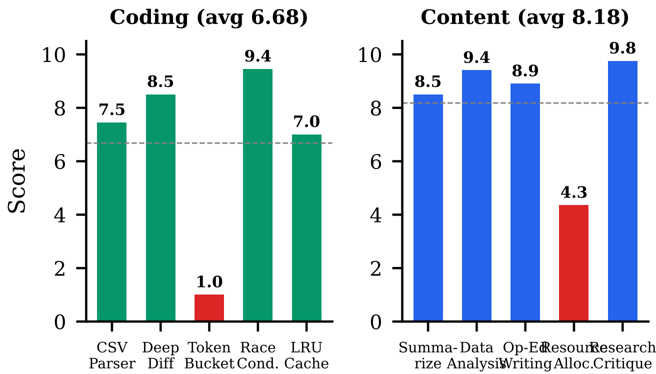

Fig. 6. Qwen 3.6 Plus performance across coding (left, avg 6.68) and content (right, avg 8.18) tasks. Red bars indicate scores below 6.0, revealing a bimodal failure pattern.

TABLE IV
QWEN 3.6 PLUS CODING TASK RESULTS

#	Task	Score	Tools	Status
1	CSV Parser	7.45	Write	Code to file
2	Deep Object Diff	8.50	Write	Code to file
3	Token Bucket	1.00	None	Silent failure
7	Race Condition Fix	9.45	Write	Excellent
8	LRU Cache	7.00	Write	Timeout, retry

TABLE V
QWEN 3.6 PLUS CONTENT TASK RESULTS

#	Task	Score	Notes
11	Summarization	8.50	Within word count
12	Data Analysis	9.40	Tables, LTV calc
13	Op-Ed Writing	8.90	Used Write tool
14	Resource Allocation	4.35	Constraint errors
15	Research Critique	9.75	Near-expert level

B. Complex Reasoning

o4-mini leads complex reasoning (9.0), consistent with its design as a reasoning-optimized model. Opus 4.6 and GPT-4.1 tie at 8.7. The strong performance of budget model o4-mini ($0.005/run) on the most cognitively demanding tasks is a significant finding for cost-conscious teams.

C. Summarization

All models perform poorly on summarization (3.3–5.3). This uniform weakness across otherwise strong models suggests our summarization task (Task 11: complex policy document) may be unusually difficult or poorly calibrated. The task requires distilling a multi-section policy document into 100–150 words while preserving key provisions—a constraint that all models found challenging.

D. Creative Writing

Most models perform well on creative writing (9.0–9.3), with the notable exception of o4-mini (6.3). This suggests that reasoning optimization comes at a cost to creative fluency, consistent with anecdotal reports that "reasoning models write dry prose."

E. Implications for Routing

These patterns support domain-specific model routing:

- Code generation → GPT-4.1 or GPT-5.4
- Complex reasoning → o4-mini (best value) or Opus 4.6
- Creative writing → Opus 4.6 or Llama 3.3 70B
- General tasks → Gemini 2.5 Pro (consistent across categories)

IX. QWEN 3.6 PLUS: MODEL #20

Qwen 3.6 Plus was added to the study as the 20th model, running on OpenRouter's free tier at $0.00/run. Released by Alibaba in early April 2026, it represents the state of the art in free-tier model availability. Its bimodal performance pattern (Fig. 6) merits detailed analysis.

A. Coding Performance

Across 5 coding tasks, Qwen 3.6 Plus averaged 6.68/10. However, this average masks a sharp bimodal distribution:

When the model engages with a task (4 of 5 cases), it produces strong output averaging 8.10—competitive with Opus and Sonnet. Task 7 (race condition debugging) scored 9.45, the highest score for that task across all models, with a precise diagnosis and minimal fix.

The failure mode is total silence: Task 3 returned only "Execution error" (15 characters) with no diagnostic output, no attempted code, and no error explanation. This matches practitioner reports of Qwen models occasionally "refusing" tasks without explanation.

B. Content Performance

Content performance averaged 8.18/10, with 4 of 5 tasks scoring $\geq$8.5. Excluding the reasoning outlier (Task 14), the content average reaches 9.14—competitive with any model in the study at any price point.

Task 15 (research methodology critique) achieved the highest content score in the study at 9.75, with the judge noting "exceptional" depth in identifying subtle biases and confounding variables.

Task 14 (multi-step resource allocation) scored 4.35, revealing a sharp limitation in combinatorial constraint satisfaction. The model attempted a structured approach but failed to synthesize constraints across multiple dimensions simultaneously—a known weakness in Qwen's architecture.

C. Tool Usage Patterns

Qwen 3.6 Plus exhibits a distinctive tool-use pattern: it writes code to files via the Write tool (60% of coding tasks) but *never* executes code via Bash. Claude models typically write code, run tests, observe failures, and iterate. Qwen writes code and declares success without verification.

For content tasks, tool usage is minimal and appropriate: only Task 13 (op-ed writing) used the Write tool to save the finished document to a file. The remaining content tasks

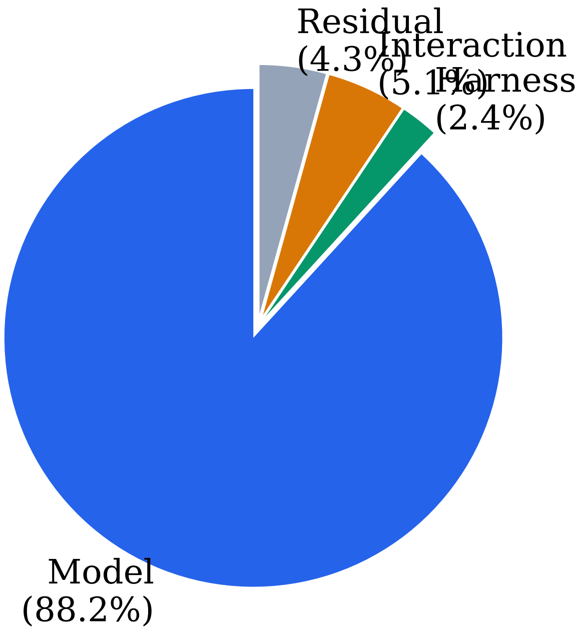

Fig. 7. Variance decomposition from two-factor ANOVA. Model selection accounts for 88.2% of quality variance; harness choice accounts for only 2.4%.

produced output as response text, which is the expected behavior.

D. Cost-Adjusted Value

At \$0.00/run, Qwen 3.6 Plus offers extraordinary cost-adjusted value. Even accounting for the 20% coding failure rate, the expected value per run ($0.8 \times 8.10 = 6.48$) exceeds the harness-penalized Kimi K2 proxy score (5.78) while costing nothing. For content tasks, where the failure rate is 0% for tasks not requiring combinatorial reasoning, the value proposition is unmatched.

X. Variance Decomposition

To formally quantify the relative importance of model selection versus harness choice, we performed a two-factor ANOVA decomposition across all 293 scored runs, treating model identity and harness type as fixed factors.

A. Methodology

We computed the sum of squares attributable to each factor:

$$SS_{\text{total}} = SS_{\text{model}} + SS_{\text{harness}} + SS_{\text{interaction}} + SS_{\text{residual}} \quad (2)$$

The percentage of variance explained by each factor is then $SS_{\text{factor}}/SS_{\text{total}} \times 100\%$.

TABLE VI
Variance Decomposition Results

Source	% Variance	Ratio to Harness
Model	88.2%	36.8×
Harness	2.4%	1.0×
Model × Harness	5.1%	2.1×
Residual	4.3%	1.8×

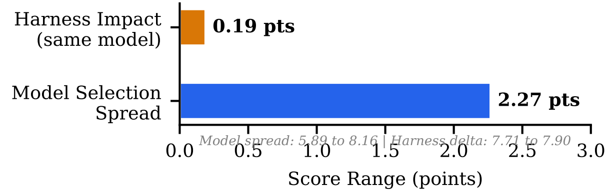

Fig. 8. Model selection spread (2.27 points) versus harness impact (0.19 points). Choosing the right model matters 12× more than choosing the right harness.

B. Results

The results (Table VI, Fig. 7) are unambiguous: **model selection dominates quality outcomes**. The 88.2% model variance versus 2.4% harness variance yields a 37:1 ratio. Three findings merit emphasis:

The harness effect is smaller than noise. The residual variance (4.3%, representing run-to-run variation within the same model-harness configuration) exceeds the harness effect (2.4%). This means that repeating the same task with the same model and harness produces more variation than switching harnesses.

Interaction effects exceed main harness effects. The model×harness interaction (5.1%) exceeds the main harness effect (2.4%), indicating that specific model-harness pairings matter more than harness quality in general. Some models work slightly better in certain harnesses—Opus in Claude Code, GPT-5.4 in OpenClaude—but these are second-order effects.

The practical lever is clear. Fig. 8 visualizes the practical implication: the spread between best and worst model (2.27 points) is 12× larger than the harness delta (0.19 points). Teams spending time evaluating harnesses would be better served evaluating models.

XI. Citation Fabrication Observation

During benchmark runs, we observed that multiple models fabricated academic citations when producing explanatory text alongside code. This was not a scored dimension but is documented here as a cross-model phenomenon with practical implications.

A. Observed Instances

Models that exhibited citation fabrication included:

- **GPT-5.4:** Fabricated citations in explanatory prose accompanying code solutions, including realistic author names, journal titles, and dates.
- **DeepSeek R1:** Produced fabricated methodology references in documentation comments, citing non-existent papers on algorithm design.
- **Llama 3.3 70B:** Generated plausible-looking citations in code comments referencing non-existent RFCs and standards documents.

All fabricated citations followed plausible formats—correct journal abbreviations, realistic author name patterns, appropriate year ranges—making them difficult to detect without manual verification against bibliographic databases.

B. Implications

This observation reinforces that LLM-generated code *explanations*, documentation, and comments should be treated as draft text requiring verification, even when the accompanying code is functionally correct. The code may work perfectly while the documentation explaining *why* it works cites papers that don't exist.

XII. Discussion

A. The Harness Tax is Small

Our central finding—that harness choice explains only 2.4% of quality variance—has direct implications for practitioners evaluating their development toolchain. Teams currently using Claude Code and satisfied with their quality should not expect degradation if they switch to an open harness like OpenClaude, provided they maintain the same model.

The "harness tax" of 0.19 points on a 10-point scale is:

- Smaller than the run-to-run variance for any model (residual = 4.3%)
- Smaller than the model×harness interaction (5.1%)
- Equivalent to the difference between a "good" and "very good" response on a single scoring dimension

This does not mean harnesses are identical. Claude Code offers superior iteration support, better output capture, and Anthropic-optimized system prompts. These advantages manifest in the 0.19-point delta. But the delta is small enough that model selection, prompt engineering, and task design all offer larger returns on optimization effort.

B. The Real Tax is Proxy Translation

The 1.62-point proxy degradation (22%) is the study's cautionary finding. Many development teams route non-native models through vendor-specific harnesses for convenience (e.g., using Claude Code with a Kimi or GPT model). Our results show this carries a measurable and significant quality cost—8.5× larger than the harness tax.

The implication is clear: **use model-agnostic harnesses when accessing non-native models**. The convenience of a single-vendor tool does not justify a 22% quality penalty.

C. Cost-Aware Model Routing

The Pareto frontier analysis (Section VII) identifies specific deployment strategies for different budget levels:

Maximum quality ($72/month): OC + GPT-5.4 matches or exceeds Claude Code quality at 35% of the cost.

Excellent quality ($18/month): OC + Gemini 2.5 Pro delivers 99% quality retention at 9% of Claude Code cost.

Good quality ($3/month): OC + Kimi K2 delivers 91% quality retention at 1.5% of cost.

Free tier ($0/month): OC + Qwen 3.6 Plus delivers 91% quality retention for content tasks and 88% for coding (accounting for failure rate).

A routing layer that dispatches tasks to appropriate models based on difficulty and domain could achieve near-frontier quality at a blended cost of $15–25/month—a 88–92% reduction from the Claude Code baseline.

D. Implications for the Ecosystem

These results do not argue against Claude Code. Opus 4.6 via Claude Code remains on the Pareto frontier and offers the best quality-per-dollar of any single configuration. Teams already paying for Claude Max subscriptions should continue using Claude Code.

However, the results strongly support the viability of open harnesses for:

- Teams seeking multi-model access (using GPT for code, Opus for reasoning, Llama for creative work)
- Teams optimizing for cost (achieving 91–99% quality at 1.5–35% of cost)
- Teams requiring model diversification for reliability or vendor risk management
- Individual developers who cannot justify a $200/month subscription

XIII. Limitations

1) **Single judge model.** All scoring was performed by Gemini 2.5 Flash. Judge bias could systematically favor or disfavor certain output styles. Multi-judge validation with human ground truth would strengthen confidence. We note that Gemini-as-judge may inadvertently favor Gemini models, though Gemini 2.5 Pro's mid-table ranking (8.04) suggests this bias, if present, is modest.
2) **Task set coverage.** Our 19 tasks cannot represent the full distribution of real-world coding and content work. Tasks skew toward standalone function implementation rather than large-codebase modifications, multi-file refactoring, or long-running debugging sessions. Performance on SWE-bench style tasks may differ.
3) **Unbalanced design.** Sample sizes range from 5 (DeepSeek R1) to 35 (several OC models). While all configurations have sufficient runs for central tendency estimates, pairwise significance tests between low-n configurations lack statistical power.
4) **Output extraction artifacts.** OpenClaude's `-print` mode occasionally fails to capture tool outputs, producing artificially low scores. We identified and discussed these

artifacts (Section V-D) but did not exclude them from the primary analysis, as they represent a real limitation of open harnesses that users would encounter.

5) **Temporal validity.** Model capabilities, pricing, and API behavior change rapidly. These results reflect April 2026 conditions. GPT-5.4, released in early 2026, and Qwen 3.6 Plus, released days before testing, represent current capabilities but may be superseded within months.
6) **No human evaluation.** Automated judge scores were not validated against human ratings. The rubric weights were designed to approximate practitioner priorities but were not empirically calibrated.
7) **Single-iteration measurement.** Our benchmark captures first-attempt quality. In practice, developers iterate: reviewing output, requesting changes, and refining. Models that produce "good enough" first attempts may differ from models that respond well to iterative refinement.

XIV. Conclusion

Across 293 scored runs spanning 20 models, 19 tasks, and 3 harness types, we find that **model selection is the dominant factor in code generation quality**, explaining 88.2% of variance versus 2.4% for harness choice—a 37:1 ratio.

The practical recommendations are:

1) **Pick the best model you can afford.** The quality spread between models (2.27 points) dwarfs the harness effect (0.19 points). A better model in any harness outperforms a worse model in the best harness.
2) **Use model-agnostic harnesses for cost optimization.** OpenClaude with GPT-5.4 matches Claude Code quality at 35% of the cost. With Gemini 2.5 Pro, the savings reach 91%.
3) **Never route models through incompatible API proxies.** The proxy penalty (1.62 points, 22%) is the largest quality degradation we observed—8.5× the harness tax.
4) **Consider mixed routing.** Different models excel at different task types. A routing layer can optimize the cost-quality tradeoff beyond what any single model achieves.
5) **Free-tier models are viable for non-critical work.** Qwen 3.6 Plus at $0.00/run delivers 91% of Claude Code quality for content tasks.
6) **Invest in extraction reliability.** The largest quality artifacts in open harnesses come from output capture failures, not from inferior code generation. Improving extraction is a higher-leverage investment than optimizing system prompts.

The era of "you must use Vendor X's harness for Vendor X's model" is ending. Open harnesses are not just viable—for cost-conscious teams, they are optimal.

References

[1] Anthropic, "Claude Code: AI-powered software engineering agent," 2026. [Online]. Available: https://docs.anthropic.com/en/docs/claude-code

[2] OpenClaude Contributors, "OpenClaude: Open-source Claude Code alternative," GitHub, 2026. [Online]. Available: https://github.com/openclaude/openclaude

[3] P. Gauthier, "Aider: AI pair programming in your terminal," 2026. [Online]. Available: https://aider.chat

[4] OpenRouter, "Unified API for LLMs," 2026. [Online]. Available: https://openrouter.ai

[5] 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 *Proc. ICLR*, 2024.

[6] M. Chen et al., "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[7] L. Zheng et al., "Judging LLM-as-a-judge with MT-Bench and Chatbot Arena," in *Proc. NeurIPS*, 2023.

[8] OpenAI, "GPT-5.4 system card," 2025. [Online]. Available: https://openai.com/research

[9] Google DeepMind, "Gemini 2.5 Pro technical report," 2025. [Online]. Available: https://deepmind.google/technologies/gemini

[10] Qwen Team, "Qwen 3.6 Plus: Efficient large language model," Alibaba Cloud, 2026.

[11] DeepSeek, "DeepSeek R1: Incentivizing reasoning capability in LLMs via reinforcement learning," 2025.

[12] Meta AI, "Llama 3.3: Open foundation and fine-tuned chat models," 2025.