SWE-QUALITY: A BENCHMARK FOR CODE QUALITY IN AGENTIC SOFTWARE ENGINEERING

Anonymous authors
Paper under double-blind review

ABSTRACT

As LLM agents increasingly automate software engineering tasks, a major bottleneck to deploying their code is **code quality**. Low-quality code is hard to read, hard to edit and expand, and may be less reusable. Repeated edits by agents over the lifetime of a repository may degrade its quality. Despite its importance, agentic benchmarks score correctness almost exclusively. We explore cheap-to-compute static metrics for evaluating agentic code quality. To that end, we introduce **SWE-Quality**, a benchmark built on DeepSWE that computes *quality scores* for agent-produced repository-level patches. SWE-Quality supplies **(i)** quality annotations for DeepSWE v1.1: 72 static metrics spanning a broad array of metric families, under 3 file-scope lenses (216 features), for 3,402 completions from 26 models, and the gold patches of its 33 Python tasks; **(ii)** a scoring protocol built for cross-model fairness, utilising within-task robust standardisation and tie-aware scoring; **(iii)** a validated scoring function, the *SWE-Quality Aligned Index*, a learned combination of those metrics that, on held-out tasks, reproduces 80% of the orderings a blind LLM judge (Claude Fable 5.1) gave to 1,204 within-task pairs of completions from 9 of DeepSWE's best performing models, and ranks those models almost exactly as the judge does (Spearman $\rho = +0.95$); and **(iv)** a baseline leaderboard with ablations. By showing that cheap, deterministic and interpretable metrics reproduce most of an independent judge's quality orderings, SWE-Quality provides a scalable instrument for agentic code quality.

1 INTRODUCTION

As LLM agents automate more software engineering tasks, the bottleneck to deploying their code is not only correctness, but quality. Low-quality code is hard to read, difficult for humans to review, hard to edit and extend, and less reusable. Agents now operate on whole repositories and are deployed to modify the same codebase repeatedly, which raises a question most current evaluations cannot answer: Will repeated agent edits degrade code quality over a repository's lifetime? Software evolution research holds that repeated human modification without attention to structure degrades a system, through rising complexity (Lehman, 1980), software aging (Parnas, 1994), measurable code decay (Eick et al., 2001) and compounding technical debt (Kruchten et al., 2012). The code an agent leaves behind must still be understood, maintained and extended by humans and agents alike. Answering that question first requires a reliable measure of the quality of a single agent change, which is what this paper builds.

Understandability, maintainability and extensibility relate naturally to observable aspects of code. Long functions and files must be read in full before they can be changed safely (Fowler, 1999). Branching and deep nesting multiply the paths through the code and the context a reader must hold (McCabe, 1976; Harrison & Magel, 1981; Campbell, 2018). Comments aid comprehension (Tenny, 1988). How functionality is divided across functions, classes, modules and files, that is, its modularity, governs how far a change ripples and how easily a feature can be added (Parnas, 1972; Stevens et al., 1974; Chidamber & Kemerer, 1994). Duplicated code must be changed consistently wherever it appears (Juergens et al., 2009). Each of these aspects can be measured statically by the classical families of software metrics, and SWE-Quality spans fourteen of them (§2).

Standard evaluations like SWE-bench (Jimenez et al., 2024) and its descendants score a single binary outcome, whether the patch makes the hidden tests pass (Yang et al., 2024; Xia et al., 2025).

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

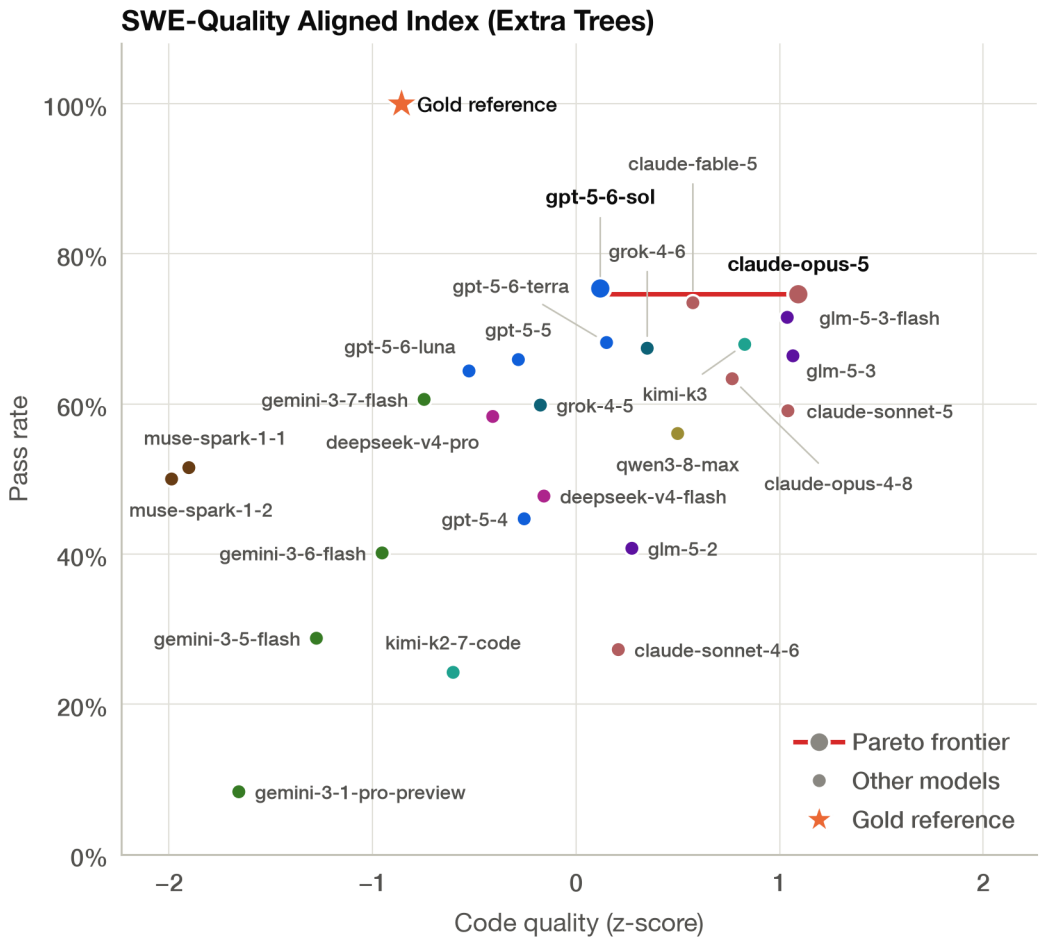


Figure 1: **The SWE-Quality Aligned Index (Extra Trees) versus correctness across 26 models.** x: the aligned index of §4.4, scored only on completions the judge never saw (z-score). y: the strict pass rate on DeepSWE's 33 Python tasks under DeepSWE's own scoring. The aligned index and *Overall* (Figure 2a) order the 26 models almost identically (Spearman $\rho = +0.97$), and the index ranks the 9 judged models almost exactly as the LLM judge does ($\rho = +0.95$, §5.1). The maintainers' reference patches are the ★. Models from the same family share a dot colour.

Benchmarks define what gets optimized, so one that scores only correctness leaves quality unmeasured and unimproved. The few evaluations that look past the tests withhold their tasks and rubrics, proxy quality by later test outcomes, track a single agent's trajectory, or apply static-analysis tools whose scores are never checked against judgement, so none gives a validated public axis for comparing models (§2). Nor does correctness stand in for quality (§5.4).

Measuring quality directly is hard. Quality is highly subjective and a reliable measure would need many human experts comparing completions from many models across many tasks. LLM judges (Zheng et al., 2023b) are more scalable, but still expensive. They are also impractical when every individual completion must be scored, as is necessary for dense rewards in reinforcement learning. Classical software complexity metrics are the opposite: deterministic, cheap enough to compute on every completion, and interpretable. Whether they are a *reliable* proxy for human-perceived quality, however, is disputed (Shepperd, 1988; Fenton & Neil, 1999). There is also no ready-made target to compare against. The obvious candidate, the maintainers' own "gold" patches, is not the best-quality code. The blind LLM judge we use for validation, Claude Fable 5.1, ranks them below most model completions, and human raters never rank them best (§5.2, §5.3). We therefore ask whether classical

complexity metrics can serve as a reliable proxy for agentic code quality, and we try to answer this question empirically.

SWE-Quality. We introduce **SWE-Quality**, a benchmark built directly on DeepSWE v1.1 (Huang et al., 2026). We add *quality annotations* to its agent-produced repository-level patches without altering DeepSWE's tasks, completions or outcomes (§3). Quality is scored as a second axis alongside correctness, which DeepSWE already provides (Figure 1). The *SWE-Quality Aligned Index* combines several algorithmically computable metrics and fits them to the rankings of a blind LLM judge (Claude Fable 5.1). Our model is based on an analysis of completions from 9 of the best performing currently available frontier models. We also check the judge against a set of human rankings and against a second LLM judge, Claude Opus 5 (§5.3). To the best of our knowledge, SWE-Quality is the only public benchmark that scores the code quality of agentic patches with a broad array of quickly computable and reproducible metrics, so any model can be ranked on quality. Evals should be cheap and easy to run by anyone. The dominant LLM-as-a-judge techniques are high variance, and thus require large sample sizes across multiple judge models to provide reliable signal(Zheng et al., 2023a; Liusie et al., 2024; Li et al., 2025). Today, doing this with frontier models for even a small subset of coding problems costs hundreds of dollars. While LLM judges were used to calibrate the SWE-Quality metric, the resulting benchmark is a powerful proxy that can be run instantly and effectively for free.

Contributions.

- **Quality annotations for DeepSWE v1.1**: 72 static metrics across fourteen families, under 3 file-scope lenses (216 features), for 3,402 completions from 26 models, each at its best performing reasoning effort, and the gold patches of 33 Python tasks, blind Claude Fable 5.1 rankings over $\mathbf{1{,}204}$ within-task ordered pairs (290 completions from 9 of the best performing models and gold) and human ranking of the strongest models for a subset of tasks (§3, §5.3).
- **A fair scoring protocol**: 3 file-scope lenses, within-task robust standardisation and tie-aware scoring (§4).
- **A validated scoring function, cheap enough to score every completion**: the SWE-Quality Aligned Index reproduces 80% of the judge's within-task orderings on held-out tasks, close to the 88.5% a second frontier judge reaches on the same pairs, and ranks the judged models almost exactly as the judge does (Spearman $\rho = +0.95$). It spends no model tokens, scoring a repository in a median of 1.35 s on one CPU core and a further patch in 313 ms (§5.1, Appendix B).
- **A baseline leaderboard with ablations**: 26 models ranked on quality apart from correctness.

2 Related Work

SWE-Quality scores patches with 72 classical software metrics from fourteen families (Table 3, Appendix A), and combines them into a single score that agrees with the far more expensive procedure of having a judge compare patches. The full definition for each metric is given in Appendix B). We review the most established metrics and their validation record, then discuss prior work on evaluation of AI-generated code.

2.1 Classical software metrics and their validation record

Size and structure. **Lines of code** is the longest-standing measure in the field, introduced as a predictor of development effort (Albrecht & Gaffney, 1983). We record it alongside **characters**, **files** and counts of declarations such as **classes** and **functions** (Lorenz & Kidd, 1994). Size carries no quality signal of its own and confounds the measures that do. We therefore keep these size metrics as context, with no better or worse direction, so that no candidate benefits purely by producing a larger patch.

Halstead's software science and the maintainability index. From counts of operators and operands, Halstead (1977) derived program **volume**, **difficulty** and **effort**, a derivation later disputed (Fitzsimmons & Love, 1978). Halstead's effort and McCabe's complexity nonetheless predicted how

long maintenance took and how accurately it was carried out (Curtis et al., 1979). The **maintainability index** combines volume, complexity, size and comment density into one figure calibrated against expert assessments (Oman & Hagemeister, 1992; Coleman et al., 1994), though it has been criticised as opaque (Heitlager et al., 2007). **Comment density** completes the family and has been examined directly in comprehension studies (Tenny, 1988).

Control flow and function shape. **Cyclomatic complexity** counts the linearly independent paths through a function (McCabe, 1976), which Shepperd (1988) argued is weakly founded and often no better than a line count. **Nesting depth** captures how many enclosing conditions and loops a reader must hold in mind at once (Harrison & Magel, 1981). **Cognitive complexity** refines it by charging an increment for each break in the linear flow and again for the depth at which the break occurs (Campbell, 2018), which correlates with comprehension time if less consistently with comprehension accuracy (Muñoz Barón et al., 2020). Long functions and long parameter lists are among the best-known code smells (Fowler, 1999), and fault density is lowest for components of intermediate size (Hatton, 1997). We record **function length** and **parameter counts** alongside the complexity measures, each as a mean and a maximum. Cognitive complexity is the strongest single metric in our validation (§5.1).

Coupling and cohesion. Coupling is the extent to which one unit of code depends on others, cohesion the extent to which the elements within a unit belong together, and structured design established low coupling and high cohesion as design goals (Stevens et al., 1974). Henry & Kafura (1981) made them countable as **fan-out:** the modules a module imports, and **fan-in:** those that import it. Chidamber & Kemerer (1994) gave them object-oriented form, **coupling between objects:** the distinct classes a class references, and **lack of cohesion:** whether its methods share attributes. The family has the most consistent fault-prediction record across systems and languages (Basili et al., 1996; Gyimóthy et al., 2005).

Duplication. Duplicated code heads the catalogue of code smells (Fowler, 1999), and inconsistent modification of clones is an established source of faults (Juergens et al., 2009). Its harmfulness is contested, since cloning is deliberate (Kapser & Godfrey, 2008) and detection tools disagree on what counts as a clone (Roy et al., 2009). We report **duplicated lines** and the **proportion of code** they represent.

Readability and documentation. Readability models are fitted to surface properties of source text and predict human judgements with useful accuracy (Buse & Weimer, 2010; Scalabrino et al., 2018), and we take those properties directly, chiefly **indentation depth** and **line length**. **Docstring coverage**.

2.2 AI-GENERATED CODE AND QUALITY-AWARE AGENTIC BENCHMARKS

Agentic benchmarks almost exclusively score correctness. SWE-bench (Jimenez et al., 2024) moved evaluation to real repository issues resolved by agents (Yang et al., 2024), and DeepSWE (Huang et al., 2026), on which we build, extends it to original long-horizon tasks in five languages. Correctness overstates usefulness. Maintainers decline about half of the agent patches that pass SWE-bench Verified's tests, and accept only about two thirds of the original human-written ones (METR, 2026).

Recent benchmarks look past the test outcome by different routes: maintainability as correctness on a repository's future commits (Chen et al., 2026) or as regression safety across chained upgrades (Lam et al., 2026), both still measured by running tests; an LLM review added to online-judge tests (Lu et al., 2026); two composite static measures along one agent's own trajectory (Orlanski et al., 2026); and, industrially, expert rubrics for whether a maintainer would merge, whose tasks, rubrics and grades are withheld to avoid contamination (Cognition, 2026). Closest to us, Chen & Jiang (2025) use SonarQube, an automated code analysis tool, to measure how 4,892 patches from ten top-ranked SWE-bench Verified agents, and the gold patches, change reliability, security and maintainability (complexity, duplication and code smells), alongside patch patterns and issue complexity. They note that such metrics may miss aspects of quality but do not test whether they *order* patches as human judgement would.

None of these approaches yields a validated, reusable quality axis. Withheld rubrics cannot be audited and cost more to grade a completion than to generate it, test-based proxies inherit the blind spot they set out to fix, two composites along one trajectory cannot rank models, and unvalidated tool scores leave open whether they track quality. SWE-Quality is public and deterministic, scores all 3,435 patches, standardises within task so that 26 models are comparable on identical work, spans 72 metrics across fourteen families (the closest prior work uses five tool measures), and, alone among them, is *validated* against blind judgement on ordering completions of the same task.

2.3 LLM JUDGES FOR CODE

LLM judges track human preference on open-ended generation (Zheng et al., 2023b), and structured prompting improves that agreement (Liu et al., 2023). For code they are now studied in their own right. Judges are benchmarked on generation and repair (Jiang et al., 2025), on rule-perturbed programs (Wang et al., 2025), on rubric verification over agent trajectories (Peng et al., 2026), and released with judge ratings along preference dimensions (Weyssow et al., 2025). It is known that judges perform best when used for relative pair-wise comparisons (Zheng et al., 2023a; Liusie et al., 2024; Li et al., 2025) rather than absolute scoring. We use a large number of relative judgments once, as the target against which efficient and reproducible metrics are calibrated, after which the metrics carry the benchmark. It should be noted that LLM judges have their biases, and these may differ from human preferences. These include position, verbosity and self-preference bias (He et al., 2025), a preference for reference-style patches (AI21 Labs, 2026), rubric-level misalignment with developers (Mittal et al., 2026), and low agreement with human readability ratings for some languages (Ouédraogo et al., 2025). We therefore hold correctness fixed, anonymise and shuffle candidates, and average over many comparatively judgements within a task (§4.3), but we acknowledge that these biases still remain, see comparisons with human judgments in §5.3.

3 THE SWE-QUALITY BENCHMARK

SWE-Quality answers a single question: *given a repository-level task and a patch that solves it, can we measure the quality of the code the agent left behind; how maintainable, extensible and understandable it is, cheaply and accurately?*

Built on DeepSWE. SWE-Quality annotates the Python tasks of DeepSWE v1.1 (Huang et al., 2026): 33 original long-horizon feature tasks drawn from active open-source repositories. Each task has a pinned base commit, an isolated environment, a hand-written verifier that accepts any implementation of the requested functionality, and a reference (*gold*) patch. DeepSWE's released trials, which are patches from the `mini-swe-agent` harness graded by each task's verifier, supply the correctness scores unchanged. The tasks are substantial feature additions (the gold patches touch 180 files across our 33 tasks). DeepSWE's verifier checks behaviour, not code structure, so nothing SWE-Quality measures is already reflected in its score. Most importantly for this work, DeepSWE's tasks are contamination-free. They were written from scratch, so no model has seen their solutions during pre-training.

We cover DeepSWE's 33 Python tasks and the 26 models whose patches are in the released archive. We evaluate each model only at its best performing reasoning effort by DeepSWE's own leaderboard pass rate, with all four attempts per task. This gives 3,402 model completions, all applying cleanly and 1,862 passing, plus the 33 gold patches. To each completion we add 72 static metrics across fourteen families under 3 file-scope lenses (§4.1), giving 216 metric–lens features, defined and computed in Appendices A–B. The LLM judge, Claude Fable 5.1, ranks 290 completions, from 9 of the best performing models and gold, yielding 1,204 within-task ordered pairs (§4.3). Every metric annotation is deterministic given DeepSWE's patches. One analyser pass yields all 216 features, taking a median of 1.35 s for a full repository on one CPU core, and re-scoring a further patch on a repository already measured takes a median of 313 ms (Appendix B).

4 METHODOLOGY

4.1 WHICH FILES TO MEASURE

The measurement scope is the benchmark's central fairness decision. Scoring the *whole repository* drowns the patch in unchanged code, so a repository-wide maximum is usually identical for every candidate and cannot discriminate. Scoring only the files *the model itself touched* is worse. A model that implements a feature in one tidy new file is then compared on a different surface from one that threads it through existing modules, which rewards avoiding the existing code. We compute three lenses and report all of them:

focus. The files *gold* changed. A model working elsewhere is measured on code it did not write.

union. Gold's files ∪ every file that at least three *distinct* models modified. This is a cross-model consensus of "the files this task is about". It admits alternative implementation sites while rejecting incidental edits. Every model is scored on the same surface.

total. The whole repository after the patch, as a robustness check.

Each metric–lens pair is a separate feature, since lenses can order candidates differently.

4.2 WITHIN-TASK ROBUST STANDARDISATION

Raw metric values are incomparable across tasks, so all analysis standardises every metric within its task, on a median and a robust spread, with lower-is-better metrics flipped so that higher always means better (Eq. 1, Appendix D). The transform preserves order within a task, so it cannot manufacture rank agreement.

Two units are standardised. A *patch* is standardised against its task's other candidates. That is the unit the judge ranks, so it is the unit the metrics are validated on and the unit every index is fitted on and applied to (§4.4). A *model* is scored on a metric by collapsing its completions on a task to their median value, standardising that across the models on that task by the same transform, and averaging over tasks (Table 2, Appendix D).

4.3 THE LLM JUDGE

To obtain a quality judgement independent of the metrics, we used a single judge agent, Claude Fable 5.1. For each task, the judge was shown the base repository, the task description and every candidate patch at once. The patches were anonymised and randomly relabelled. The judge was asked to rank them and to give each one a score in $[0, 100]$ that weighs maintainability, extensibility and understandability. Appendix C gives the full prompt. Presenting all candidates in one context makes the judgement comparative, the regime in which judges are most reliable (Zheng et al., 2023b).

We judged a passing completion per model per task, the critical design choice. **Correctness is held constant**, so the judge compares implementations that all satisfy the same tests and cannot reward a patch for merely working. The judge returns a complete ranking of the candidates in each task, so the effective unit of evidence is the *within-task ordered pair*. The design yields $\mathbf{1{,}204}$ such pairs from 290 completions over 33 tasks, from 9 of the best performing models on DeepSWE and gold (2–10 candidates per task, median 10). We judge only these leading models because they are the ones a practitioner would realistically deploy and their completions are the hardest to tell apart. Two tasks have only one passing model besides gold, too few for a within-task correlation, so alignment to metrics is measured on the other 31.

4.4 MEASURING ALIGNMENT AND COMBINING METRICS

Judge scores are subjective and comparable only within a task, so we evaluate agreement on **order**. For each task we correlate a signal with the judge's scores across that task's candidates and average over tasks. A single metric is never fitted but oriented by its literature direction, so its *aligned* Spearman ρ (Spearman, 1904) is positive when it agrees with the judge. The headline measure is **concordance**, the share of within-task pairs a signal orders the way the judge did, a rescaling of

Table 1: **Alignment with judged quality on held-out tasks.** *Top*: the five single metrics with the highest concordance, each at its best lens and oriented by its literature direction, unfitted. *Middle*: *Overall* (Figure 2a), scored the same way. *Bottom*: combined indices fitted on the 215 usable features, with the number each uses. ⋆The SWE-Quality Aligned Index, the benchmark's main score. Every row is scored on the test tasks of the same repeated $80/20$ splits, and $\pm$ is a standard error clustered by task (Appendix D). Table 12 has all 147 directional combinations with their directions.

Signal	Lens / method	Aligned ρ	Concordance
`avg_cognitive_complexity`	union	$\mathbf{+0.651 \pm 0.047}$	$\mathbf{0.766 \pm 0.020}$
`avg_indentation`	union	$+0.508 \pm 0.041$	0.696 ± 0.018
`docstring_coverage`	focus	$+0.410 \pm 0.068$	0.665 ± 0.026
`duplication_ratio`	total	$+0.374 \pm 0.071$	0.648 ± 0.029
`avg_max_nesting`	union	$+0.341 \pm 0.056$	0.641 ± 0.023
Overall	all 216 features, unweighted	$+0.591 \pm 0.041$	0.733 ± 0.019
Combined index⋆	Extra Trees, 206 features	$\mathbf{+0.725 \pm 0.037}$	$\mathbf{0.797 \pm 0.018}$
Combined index	XGBoost, 139 features	$+0.709 \pm 0.035$	0.787 ± 0.017
Combined index	LASSO (L1), 22 features	$+0.652 \pm 0.042$	0.764 ± 0.019

Kendall's tie-corrected τ_b (Kendall, 1938) onto $[0, 1]$ where 0.5 is chance (Appendix D). A task on which a signal is constant orders nothing and is scored at that baseline, making this score **tie aware**.

We also fit combined indices on 215 of the 216 features, dropping the focus-lens test-file count, which is identical for every candidate in a task. We fit three kinds of index. The first is linear, with a LASSO penalty (Tibshirani, 1996) for the strongly collinear features (Appendix D). The other two are tree ensembles. **XGBoost** (Chen & Guestrin, 2016) fits gradient-boosted trees in sequence under a squared-error objective. **Extra Trees**, extremely randomised trees (Geurts et al., 2006), instead average many trees grown independently, each offered a small random share of the features and splitting at a random threshold. With 26 training tasks, variance rather than expressiveness is plausibly the binding constraint, which would favour averaging decorrelated trees over boosting them. Extra Trees is the most accurate index we find (§5.1), and the one the benchmark uses.

The fitting protocol. Every method is fitted and tuned by one rule, so none is given an advantage the others are denied. We draw 300 repeated $80/20$ splits of whole *tasks*, never completions, and inside each split's 26 training tasks choose every hyper-parameter by **grouped 4-fold cross-validation** over whole tasks, scored on concordance itself. The chosen model is refitted on the training tasks and scored once on the test tasks, and no judge label from those tasks touches the fit, so every effect size is held out (Appendix D).

These splits only *evaluate* alignment. The benchmark's main score, the **SWE-Quality Aligned Index**, is one extra-trees fit on all judged completions applied to the unjudged ones (Table 2, Figure 1).

5 Benchmark Results

5.1 Validating the scoring function against blind judgement

We test the metrics' validity (§2.1) on the 1,204 within-task ordered pairs of the judge's rankings (Table 1). The best single metric is cognitive complexity, which orders 77% of held-out pairs like the judge (aligned $\rho = +0.65$), with indentation (70%) and docstring coverage (66%) next and duplication ratio at 65%, while *Overall* (Figure 2a) orders 73%. Forty-two of the 147 directional combinations agree with the judge at a false-discovery rate (FDR) of $q < 0.05$, drawn from nine of the twelve families that carry a direction. Another twelve are significant with the opposite sign, mostly size counts that grow with the patch (Appendix H).

Combining the metrics helps, but one metric dominates. The SWE-Quality Aligned Index orders 80% of held-out pairs like the judge (aligned $\rho = +0.72$), against 79% for XGBoost, while the sparse linear fit and cognitive complexity alone are level (Table 1). A second frontier judge reached 88.5% agreement on the same pairs (§5.3).

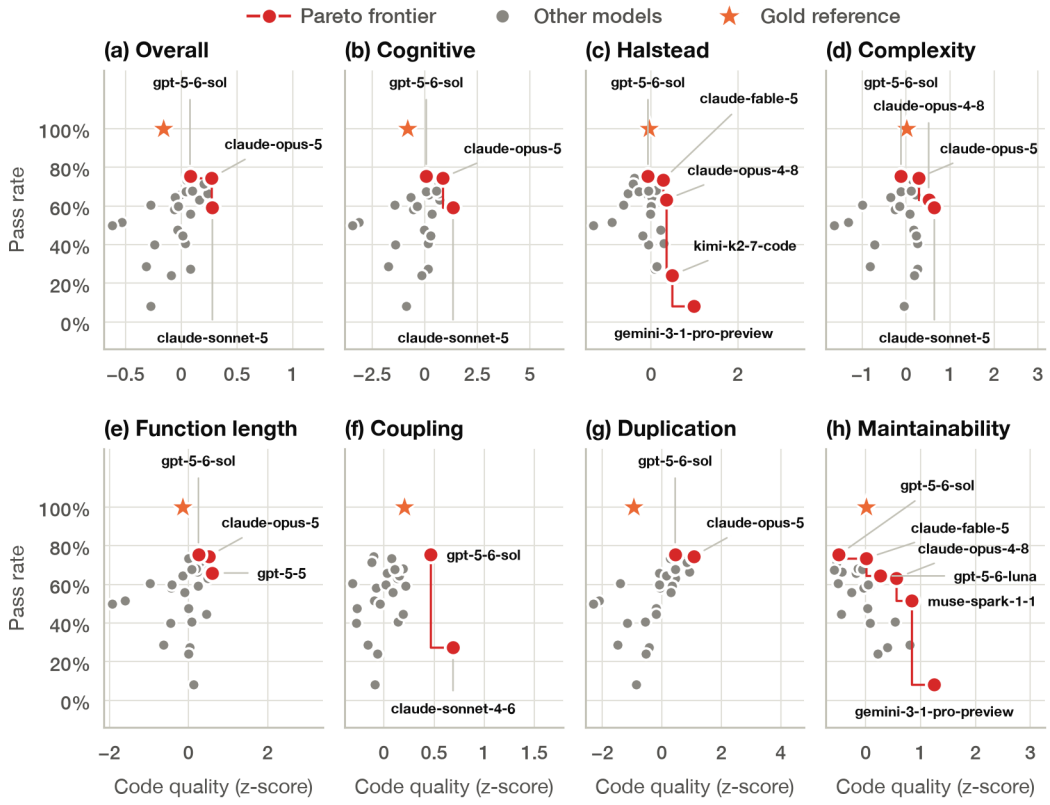


Figure 2: **Quality versus correctness overall and by metric family. (a)** *Overall*, the unweighted average of all 216 features. **(b)–(h)** Cognitive complexity, Halstead difficulty, cyclomatic complexity, function length, coupling between objects, duplication ratio and the maintainability index, the most established directional metric of seven families (Table 3), each averaged over the lenses. x: quality z, higher = cleaner. y: pass rate on the 33 Python tasks under DeepSWE's scoring. Frontier models are named.

The model ranking nearly matches the judge's. The aligned index ranks the 9 judged models almost exactly as the judge does (Table 2, Spearman $\rho = +0.95$, $p < 10^{-4}$, and $+0.96$ with gold), and this is not an artefact of fitting on the same tasks. Refitted 33 times, each without one task's labels, the index agrees with the judge just as well ($+0.95$) and ranks all 26 models as Table 11 does ($\rho = +0.999$, Appendix D). Nor are the unjudged models extrapolation. A second judge reproduces the index's order on the best model of each family (Appendix G).

5.2 The leaderboard: quality as a second axis

Figure 1 places each model on both axes, quality and correctness, which SWE-Quality reports side by side rather than combining. The Pareto frontier is `gpt-5-6-sol` and `claude-opus-5` under the aligned index. Figure 2 repeats the view for *Overall* and for one metric of each of seven families.

The maintainers' reference is not the cleanest code. The gold patches are written by the repositories' maintainers. Yet on the aligned index they score -0.86, below 21 of the 26 models, and the judge ranks gold last of 10 on average (Table 2), so it cannot serve as the quality target.

5.3 Agreement with a second judge and human raters

The judge is itself a proxy, so we checked it twice. First, Claude Opus 5 re-judged the same blinded candidates (§4.3). The two judges order the nine models and gold identically (Spearman $\rho = +1.00$), agree on 88.5% of the 1,204 within-task pairs and pick the same best candidate in 20 of the 33 tasks.

Table 2: **The judged models, 9 of the best performing on DeepSWE, ranked by the SWE-Quality Aligned Index.** *Quality*: z-scores for the aligned index (Extra Trees), scored only on unjudged completions (§4.4), and for the best single metric, cognitive complexity on the union lens (Table 1). *LLM judge*: Claude Fable 5.1's mean score, mean normalised rank (0 = best in task) and tasks ranked first. *Correctness*: the strict pass rate on DeepSWE's 33 Python tasks under DeepSWE's scoring. †Gold, whose only patch was judged, is scored by a refit without the gold rows.

Model	Quality (z)		LLM judge (Claude Fable 5.1)			Correctness
	Aligned index	Cognitive	Score	Norm. rank ↓	Wins	Pass rate (%)
`claude-opus-5`	$+1.09$	$+0.96$	76.4	0.23	8	74.6
`glm-5-3`	$+1.07$	$+0.82$	77.3	0.18	9	66.4
`kimi-k3`	$+0.83$	$+0.70$	74.8	0.27	4	67.9
`claude-fable-5`	$+0.57$	$+0.57$	75.0	0.25	6	73.5
`grok-4-6`	$+0.35$	$+0.19$	65.8	0.50	2	67.4
`gpt-5-6-sol`	$+0.12$	$+0.03$	68.4	0.41	4	75.4
`gpt-5-5`	-0.28	$+0.01$	58.8	0.64	0	65.9
`gpt-5-6-luna`	-0.53	-0.75	56.2	0.68	0	64.4
`gemini-3-7-flash`	-0.75	-1.07	47.1	0.83	0	60.6
gold (reference)	$\mathbf{-0.86}^{\dagger}$	$\mathbf{-1.04}$	**37.0**	**0.94**	**0**	**100.0**

Second, four human raters (paper authors) ranked eight candidates (the top performing seven models and gold) on each of eight tasks in a preliminary study, giving 224 within-task ordered pairs. Claude Fable 5.1, judging the same candidates blind, orders 65% of the raters' pairs the same way (chance 50%). The raters were not given the judge's prompt and ranked by their own notion of quality, which may account for part of that gap. The raters rank the maintainers' gold patch first on none of the eight tasks, and the judge ranks it last on all eight. We see that human scores do correlate with quality metrics, but the limited sample size makes the standard errors too high to accurately fit and evaluate a combined index (Appendix F).

5.4 Negative control: the benchmark is not measuring correctness

While, many metrics are closely aligned with the judge's rankings for code quality, no metric can separate models' passing from failing completions at any statistical significance. The highest within-task AUC for any feature (equivalent to concordance for binary outcomes) with correctness is 0.577. Furthermore, all features have a q value above 0.99 against chance after FDR correction (Appendix E). Therefore, none of the classical software quality metrics we investigate in this benchmark measures correctness.

6 Limitations and Conclusion

While SWE-Quality Aligned Index enables fast a reproducible scoring of coding agents, its main limitation is the validation reference, one LLM judge cross-checked by a second judge and a small human study. Agreement with human maintainers may not hold on tasks outside the DeepSWE environments tested. The benchmark scores reported for frontier models also cover only Python tasks.

This protocol was built to enable two further directions. The first is long-horizon repository evolution. Our results do not establish that agents degrade repositories over time, since each completion is a single change to a clean base, whereas the compounding decay of software evolution (Lehman, 1980; Parnas, 1994; Eick et al., 2001) operates over many changes, and measuring the same features along chains of agent commits would settle it. The second is training rather than evaluation. A judge cannot score millions of rollouts, but a deterministic index can, at 313 ms per patch (Appendix B), which makes quality usable as a *dense reward* beside today's sparse test-based reward, though optimising a metric directly invites reward hacking, so its alignment would need re-establishing under training pressure.

AI USE STATEMENT

In this work, we used generative AI tools to implement methods, namely the metric computations, alignment experiments and the statistical tests. We have not used generative AI tools to help develop theoretical models or conceptual frameworks, to design research methodology or experiments or to provide feedback on them, to clean and reformat the dataset, to support qualitative and thematic data analysis, or to interpret results, and generating synthetic data sets, formulating mathematical claims, providing critical ingredients for proving mathematical claims, assisting in the writing of proofs, proposing or refining hypotheses, and translation are not applicable to this work.

Additionally, we used generative AI tools to create and modify scientific figures, to suggest experimental parameters, to create and edit software code, to draft parts of this paper, to summarise and analyse existing literature, to edit the paper for readability, to identify relevant literature and to format references.

The authors reviewed all AI-assisted work, checking AI-written text and code for correctness, logical consistency and scientific accuracy. We take responsibility for the final content of this paper, including any text, claims or artefacts produced with the aid of generative AI.

Language models also appear in this work as objects of study and as part of the evaluation protocol, as the completions under measurement and as the blind judges of §4.3 and §5.3. That use is part of the method and is described there.

REPRODUCIBILITY STATEMENT

Every number in this paper is deterministic given DeepSWE v1.1's released completions, which we take unchanged along with its correctness scores (§3). Appendix A lists the 72 metrics by family with the better-direction each carries, and Appendix B specifies all of them exactly as the analyser implements them, including file discovery, the three lens file sets, the formula or walk behind each metric, how a lens aggregates it, and a worked example with the values the analyser returns for it. The judge round can be repeated from the prompt, the folder layout and the candidate selection in Appendix C. Appendix D defines every statistic we report, the within-task standardisation of Eq. 1, the five-step fitting and evaluation protocol with its split counts and hyper-parameter grids, and how a model's score is built from its patch scores. Appendix F reports the second judge and the human comparison and Appendix H gives the per-model and per-metric tables behind every figure in the main text.

REFERENCES

AI21 Labs. Gold-like answers reveal llm judge bias in coding benchmarks. `https://www.ai21.com/blog/gold-like-answers-benchmarks/`, 2026.

Allan J. Albrecht and John E. Gaffney. Software function, source lines of code, and development effort prediction: A software science validation. *IEEE Transactions on Software Engineering*, SE-9(6):639–648, 1983.

Victor R. Basili, Lionel C. Briand, and Walcelio L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, 1996. doi: 10.1109/32.544352.

Yoav Benjamini and Yosef Hochberg. Controlling the false discovery rate: A practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society: Series B*, 57(1):289–300, 1995.

Raymond P. L. Buse and Westley R. Weimer. Learning a metric for code readability. *IEEE Transactions on Software Engineering*, 36(4):546–558, 2010. doi: 10.1109/TSE.2009.70.

G. Ann Campbell. Cognitive complexity: An overview and evaluation. In *Proceedings of the International Conference on Technical Debt (TechDebt)*, pp. 57–58, 2018. doi: 10.1145/3194164.3194186.

Jialong Chen, Xander Xu, Hu Wei, et al. Swe-ci: Evaluating agent capabilities in maintaining codebases via continuous integration. *arXiv preprint arXiv:2603.03823*, 2026.

Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 785–794, 2016. doi: 10.1145/2939672.2939785.

Zhi Chen and Lingxiao Jiang. Evaluating software development agents: Patch patterns, code quality, and issue complexity in real-world github scenarios. In *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 657–668, 2025. doi: 10.1109/SANER64311.2025.00068.

Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994. doi: 10.1109/32.295895.

Cognition. Introducing frontiercode. `https://cognition.com/blog/frontier-code`, 2026.

Jacob Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum Associates, 2nd edition, 1988.

Don Coleman, Dan Ash, Bruce Lowther, and Paul Oman. Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49, 1994. doi: 10.1109/2.303623.

Ward Cunningham. The wycash portfolio management system. In *OOPSLA '92 Experience Report*, 1992. doi: 10.1145/157709.157715.

Bill Curtis, Sylvia B. Sheppard, Phil Milliman, M. A. Borst, and Tom Love. Measuring the psychological complexity of software maintenance tasks with the halstead and mccabe metrics. *IEEE Transactions on Software Engineering*, SE-5(2):96–104, 1979. doi: 10.1109/TSE.1979.234165.

Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, pp. 109–118, 1999.

Stephen G. Eick, Todd L. Graves, Alan F. Karr, J. S. Marron, and Audris Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, 2001. doi: 10.1109/32.895984.

Norman E. Fenton and Martin Neil. A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25(5):675–689, 1999. doi: 10.1109/32.815326.

Ann Fitzsimmons and Tom Love. A review and evaluation of software science. *ACM Computing Surveys*, 10(1):3–18, 1978.

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

Jerome Friedman, Trevor Hastie, and Robert Tibshirani. Regularization paths for generalized linear models via coordinate descent. *Journal of Statistical Software*, 33(1):1–22, 2010.

Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Machine Learning*, 63(1):3–42, 2006.

David Goodger and Guido van Rossum. Pep 257 – docstring conventions. Python Enhancement Proposal 257, 2001. URL `https://peps.python.org/pep-0257/`.

Tibor Gyimóthy, Rudolf Ferenc, and István Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10): 897–910, 2005. doi: 10.1109/TSE.2005.112.

Maurice H. Halstead. *Elements of Software Science*. Elsevier North-Holland, 1977.

James A. Hanley and Barbara J. McNeil. The meaning and use of the area under a receiver operating characteristic (ROC) curve. *Radiology*, 143(1):29–36, 1982.

Warren A. Harrison and Kenneth I. Magel. A complexity measure based on nesting level. *ACM SIGPLAN Notices*, 16(3):63–74, 1981.

Les Hatton. Reexamining the fault density–component size connection. *IEEE Software*, 14(2):89–97, 1997.

Junda He, Jieke Shi, Terry Yue Zhuo, et al. Llm-as-a-judge for software engineering: Literature review, vision, and the road ahead. *arXiv preprint arXiv:2510.24367*, 2025.

Ilja Heitlager, Tobias Kuipers, and Joost Visser. A practical model for measuring maintainability. In *Proceedings of the International Conference on the Quality of Information and Communications Technology (QUATIC)*, pp. 30–39, 2007.

Sallie Henry and Dennis Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, SE-7(5):510–518, 1981.

Martin Hitz and Behzad Montazeri. Measuring coupling and cohesion in object-oriented systems. In *Proceedings of the International Symposium on Applied Corporate Computing*, 1995.

Wenqi Huang, Charley Lee, Leonard Tng, and Serena Ge. DeepSWE: Measuring frontier coding agents on original, long-horizon engineering tasks. *arXiv preprint arXiv:2607.07946*, 2026. We use benchmark version 1.1; tasks and verifiers: `https://github.com/datacurve-ai/deep-swe`.

Hongchao Jiang, Yiming Chen, Yushi Cao, et al. Codejudgebench: Benchmarking llm-as-a-judge for coding tasks. *arXiv preprint arXiv:2507.10535*, 2025.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations (ICLR)*, 2024.

Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 485–495, 2009.

Cory J. Kapser and Michael W. Godfrey. "Cloning considered harmful" considered harmful: Patterns of cloning in software. *Empirical Software Engineering*, 13(6):645–692, 2008.

Maurice G. Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2):81–93, 1938.

Philippe Kruchten, Robert L. Nord, and Ipek Ozkaya. Technical debt: From metaphor to theory and practice. *IEEE Software*, 29(6):18–21, 2012. doi: 10.1109/MS.2012.167.

Man Ho Lam, Chaozheng Wang, Hange Liu, et al. Swe-chain: Benchmarking coding agents on chained release-level package upgrades. *arXiv preprint arXiv:2605.14415*, 2026.

Meir M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68 (9):1060–1076, 1980. doi: 10.1109/PROC.1980.11805.

Bowen Li, Wenhan Wu, Ziwei Tang, Lin Shi, John Yang, Jinyang Li, Shunyu Yao, Chen Qian, Binyuan Hui, Qicheng Zhang, et al. Prompting large language models to tackle the full software development lifecycle: A case study. In *Proceedings of the 31st International Conference on Computational Linguistics*, pp. 7511–7531, 2025.

Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. G-Eval: NLG evaluation using GPT-4 with better human alignment. In *Proc. EMNLP*, pp. 2511–2522, 2023.

Adian Liusie, Potsawee Manakul, and Mark Gales. Llm comparative assessment: Zero-shot nlg evaluation through pairwise comparisons using large language models. In *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 139–151, 2024.

Mark Lorenz and Jeff Kidd. *Object-Oriented Software Metrics: A Practical Guide*. Prentice Hall, 1994.

Pengrui Lu, Shiqi Zhang, Yunzhong Hou, et al. Projdevbench: Benchmarking ai coding agents on end-to-end project development. *arXiv preprint arXiv:2602.01655*, 2026.

Henry B. Mann and Donald R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The Annals of Mathematical Statistics*, 18(1):50–60, 1947.

Robert C. Martin. Oo design quality metrics: An analysis of dependencies. Technical report, Object Mentor, 1994.

Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4): 308–320, 1976. doi: 10.1109/TSE.1976.233837.

METR. Many swe-bench-passing prs would not be merged into main. `https://metr.org/notes/2026-03-10-many-swe-bench-passing-prs-would-not-be-merged-into-main/`, 2026.

Microsoft. How code metrics help identify risks. Visual Studio documentation, `https://learn.microsoft.com/en-us/visualstudio/code-quality/code-metrics-values`, 2025.

Aditya Mittal, Ryan Shar, Zichu Wu, et al. Comparing developer and llm biases in code evaluation. *arXiv preprint arXiv:2603.24586*, 2026.

Marvin Muñoz Barón, Marvin Wyrich, and Stefan Wagner. An empirical validation of cognitive complexity as a measure of source code understandability. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 1–12, 2020. doi: 10.1145/3382494.3410636.

Paul Oman and Jack Hagemeister. Metrics for assessing a software system's maintainability. In *Proc. Conference on Software Maintenance (ICSM)*, pp. 337–344, 1992. doi: 10.1109/ICSM.1992.242525.

Gabriel Orlanski, Devjeet Roy, Alexander Yun, et al. Slopcodebench: Benchmarking how coding agents degrade over long-horizon iterative tasks. *arXiv preprint arXiv:2603.24755*, 2026.

Wendkûuni C. Ouédraogo, Yinghua Li, Xueqi Dang, et al. Human-aligned code readability assessment with large language models. *arXiv preprint arXiv:2510.16579*, 2025.

Robert E. Park. Software size measurement: A framework for counting source statements. Technical Report CMU/SEI-92-TR-020, Software Engineering Institute, Carnegie Mellon University, 1992.

David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

David Lorge Parnas. Software aging. In *Proc. 16th International Conference on Software Engineering (ICSE)*, pp. 279–287, 1994. doi: 10.1109/ICSE.1994.296790.

Yangda Peng, Yunjia Qi, Haotian Xia, et al. Can llm-as-a-judge reliably verify rubrics in agentic scenarios? *arXiv preprint arXiv:2606.29920*, 2026.

Peter J. Rousseeuw and Christophe Croux. Alternatives to the median absolute deviation. *Journal of the American Statistical Association*, 88(424):1273–1283, 1993.

Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7): 470–495, 2009.

Simone Scalabrino, Mario Linares-Vásquez, Rocco Oliveto, and Denys Poshyvanyk. A comprehensive model for code readability. *Journal of Software: Evolution and Process*, 30(6):e1958, 2018. doi: 10.1002/smr.1958.

Martin Shepperd. A critique of cyclomatic complexity as a software metric. *Software Engineering Journal*, 3(2):30–36, 1988. doi: 10.1049/sej.1988.0003.

Charles Spearman. The proof and measurement of association between two things. *The American Journal of Psychology*, 15(1):72–101, 1904.

Wayne P. Stevens, Glenford J. Myers, and Larry L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.

Ted Tenny. Program readability: Procedures versus comments. *IEEE Transactions on Software Engineering*, 14(9):1271–1279, 1988.

Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B*, 58(1):267–288, 1996.

Guido van Rossum, Barry Warsaw, and Nick Coghlan. Pep 8 – style guide for python code. Python Enhancement Proposal 8, 2001. URL `https://peps.python.org/pep-0008/`.

Guido van Rossum, Jukka Lehtosalo, and Łukasz Langa. Pep 484 – type hints. Python Enhancement Proposal 484, 2014. URL `https://peps.python.org/pep-0484/`.

Ruiqi Wang, Xinchen Wang, Cuiyun Gao, et al. Axiom: Benchmarking llm-as-a-judge for code via rule-based perturbation and multisource quality calibration. *arXiv preprint arXiv:2512.20159*, 2025.

Martin Weyssow, Aton Kamanda, Xin Zhou, et al. Codeultrafeedback: An llm-as-a-judge dataset for aligning large language models to coding preferences. *ACM Transactions on Software Engineering and Methodology*, 2025. doi: 10.1145/3736407.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Demystifying LLM-based software engineering agents. *Proceedings of the ACM on Software Engineering*, 2(FSE), 2025. doi: 10.1145/3715754. Article FSE037.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in neural information processing systems*, 36:46595–46623, 2023a.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, et al. Judging LLM-as-a-judge with MT-Bench and Chatbot Arena. In *Advances in Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*, 2023b.

A Metric Definitions

All metrics are computed by a deterministic static analyser over the post-patch repository tree, restricted to the files selected by the lens (§4.1). Appendix B specifies every computation exactly. Each metric carries a *better-direction*: ↓ lower is better, ↑ higher is better, or · context/size with no intrinsic direction. Directional metrics are oriented by Eq. 1 so that a higher z always means better. Context metrics are left unoriented, so for them a positive association simply means *larger*.

Table 3: **The 72 static metrics by family.** Every family is reviewed in §2, and Appendix B gives its exact computation.

Family	Metrics	Construct measured
Size / volume (·)	`physical_lines`, `code_lines`, `logical_lines`, `blank_lines`, `comment_lines`, `code_chars`, `word_count`, `files`, `python_files`, `test_files`	How much code there is. Retained explicitly so the size confound is visible (Fenton & Neil, 1999).

Table 3 continued

Family	Metrics	Construct measured
Halstead (Halstead, 1977)	`halstead_length`, `halstead_volume`, `halstead_difficulty` (↓), `halstead_effort` (↓), `halstead_bugs` (↓)	Operator and operand counts, proxies for reading volume and expression tangling.
Control flow (McCabe, 1976)	`branches` (↓), `loops` (↓), `nested_loops` (↓), `max_nesting` (↓), `avg_max_nesting` (↓), `avg_complexity` (↓), `max_complexity` (↓)	Path count and nesting depth, that is, how many ways through and how much context a reader must hold.
Function shape	`functions`, `avg_function_length` (↓), `median_function_length` (↓), `max_function_length` (↓), `avg_function_statements` (↓), `max_function_statements` (↓), `avg_params` (↓)	Unit size and interface width.
Coupling & cohesion (Chidamber & Kemerer, 1994)	`avg_cbo` (↓), `max_cbo` (↓), `avg_lcom4` (↓), `max_lcom4` (↓), `non_cohesive_classes` (↓), `avg_fan_in`, `max_fan_in`, `avg_fan_out` (↓), `max_fan_out` (↓)	Inter-module dependence and intra-class coherence, with the strongest fault-prediction record (Basili et al., 1996; Gyimóthy et al., 2005).
Duplication	`duplicated_code_lines` (↓), `duplication_ratio` (↓), `clone_clusters` (↓)	Copy-paste within the measured scope, the usual concrete form of technical debt (Cunningham, 1992).
Maintainability (Oman & Hagemeister, 1992; Coleman et al., 1994)	`avg_maintainability_index` (↑), `min_maintainability_index` (↑), `comment_ratio` (↑)	Composite maintainability estimate and comment density.
Structure (·)	`classes`, `classes_measured`, `imports`	Declaration counts.
Cognitive complexity (Campbell, 2018)	`avg_cognitive_complexity` (↓), `max_cognitive_complexity` (↓)	Flow breaks weighted by how deeply they nest, a reading-effort measure rather than a path count.
Inheritance and response (Chidamber & Kemerer, 1994)	`avg_dit` (↓), `max_dit` (↓), `avg_noc`, `max_noc`, `avg_rfc` (↓), `max_rfc` (↓), `avg_wmc` (↓), `max_wmc` (↓)	Hierarchy depth, subclass count, response set and weighted methods, the Chidamber–Kemerer metrics beside the coupling row above.
Documentation	`docstring_coverage` (↑), `type_annotation_coverage` (↑)	Share of definitions carrying a docstring (Goodger & van Rossum, 2001) and of parameters and returns carrying a type annotation (van Rossum et al., 2014). Docstrings count as code in `comment_ratio`, so this is the only measure of them.
Readability (Buse & Weimer, 2010)	`avg_line_length` (↓), `max_line_length` (↓), `avg_indentation` (↓), `blank_line_ratio` (↑), `avg_identifier_length`, `single_char_identifier_ratio` (↓), `naming_convention_ratio` (↑)	The surface features readability models are built from, plus conformance to the language's naming conventions (van Rossum et al., 2001).

Table 3 continued

Family	Metrics	Construct measured
Package dependencies (Martin, 1994)	`avg_instability`, `abstractness`, `distance_main_sequence` (↓), `cyclic_module_ratio` (↓)	Efferent and afferent coupling as instability, the abstract share, distance from the main sequence and modules caught in an import cycle.
Exception handling	`bare_except_ratio` (↓), `broad_except_ratio` (↓)	Handlers that catch everything, which van Rossum et al. (2001) advises against.

Each of the 72 metrics is computed under each of the three lenses, giving 216 metric–lens features per completion. Of these, 215 have usable within-task variance and are used in the combined models.

B How Each Metric Is Computed

This appendix specifies all 72 metrics exactly as the SWE-Quality analyser implements them, so the benchmark can be re-implemented from the paper alone. The analyser is a deterministic, standard-library Python program that reads source text and Python parse trees. It never executes the code under analysis.

From a completion to 216 features. Every completion is measured on the full post-patch repository, and each lens then aggregates the same per-file measurements over its own file set:

```
for each completion (task t, patch p):
    check out the base commit of t and apply p with 'git apply'
    R <- analyse(repository)            # every code file, once
    for L in {focus, union, total}:
        x[L] <- aggregate(R, files(L))  # 72 values per lens
```

All 3,402 released model patches and all 33 gold patches applied cleanly, so every one is analysed. The lens file sets (§4.1) are defined as follows. *focus*: every path named in the gold patch's diff headers, test files included. *union*: the focus paths plus every path modified by at least three distinct models, where a model modifies a path if any of its sample patches for the task names it. *total*: every analysed file. A listed path that does not exist after the patch (for example, a deleted file) contributes nothing. Duplication, coupling between objects (CBO) and fan-in/fan-out are computed on the whole repository and only then restricted to the lens.

Cost. One pass over the post-patch repository yields all 216 features, so measuring a completion costs one such pass. The analyser walks the repository once, reads each file once and parses each Python file once, handing the same parse tree to every family. On the 33 tasks' base repositories, which hold between 26 and 2,448 Python files (2,816 to 262,840 lines of Python), a pass takes a median of 1.35 s on one core of an Apple M4 Pro, between 0.8 and 3.5 s for the middle half of them and 19.5 s for the largest. The three lenses are almost free, as they aggregate the same per-file measurements over different file sets. Measuring all 3,435 patches costs a few core-hours and no model tokens. Timings are the fastest of three passes after an untimed warm-up.

Scoring a *further* patch on a repository already measured costs far less, which matters if the index is used as a dense reward rather than for a one-off evaluation. Most of a pass is per-file work, so the files a patch does not touch carry over and only the repository-wide steps are redone, namely duplication, coupling, the inheritance tree and the import graph. Taking one passing completion per task, a patch touches a median of 7 files and re-scoring it takes a median of 313 ms, between 150 and 504 ms for the middle half, a median 5× saving over the full pass. Every other figure in this appendix is a full pass, the pessimistic case.

File discovery. The walk skips version-control, virtual-environment, cache, build and vendored directories (`.git`, `.venv`, `venv`, `__pycache__`, `build`, `dist`, `site-packages`,

node_modules, vendor, third_party, among others), files without an extension, binary and asset extensions, and documentation, data and markup extensions (.md, .rst, .txt, .json, .yaml, .toml, .ini, .cfg, .csv, .xml, .html, .css, ...). A file whose first 4 KiB contain a NUL byte is treated as binary. The rest are decoded as UTF-8. Files ending in .py, .pyi or .pyw are parsed with Python's ast module, and a file that fails to parse is excluded from every metric. Test files are *not* excluded: test_files counts files whose name starts with test_ or ends with _test.py (or the JavaScript, TypeScript and Go test suffixes), or that lie under a tests/ or test/ directory. Other code files are measured by the lexical scanner described at the end of this appendix. They are essentially absent from the focus and union lenses (fewer than 1% of completions have one in the focus lens, 4% in the union lens), while the total lens contains a median of 4 among 192 files.

Lexical accounting (all files). A single character pass labels every character as code, string or comment. In Python, # opens a comment that runs to the end of the line, and strings are delimited by """, ''', " or ' with backslash escapes, so docstrings are string literals and count as code. A line is then a *code line* if it holds any non-whitespace code or string character, a *comment line* if it holds comment text but no code, and a *blank line* otherwise. physical_lines counts all lines. A trailing comment on a code line therefore counts as code. Following standard counting practice (Park, 1992):

- comment_ratio = comment_lines / (comment_lines + code_lines), the comment density that comprehension studies examine (Tenny, 1988);
- word_count = number of identifier-like tokens ([A-Za-z_][A-Za-z0-9_]*, keywords included) in the code text after comments and string contents are removed;
- code_chars = number of non-whitespace code characters, each string literal counting as a single character.

Python structure (parse tree). For Python files the structural counts come from the parse tree. functions counts FunctionDef and AsyncFunctionDef nodes, including methods and nested functions but not lambdas; classes counts ClassDef nodes; imports counts import/from statements (not imported names); loops counts for/async for/while statements (comprehensions excluded); and logical_lines counts statement nodes, the logical source-line count of Park (1992). branches is the number of decision points in the whole module, counted by the rule used for complexity below, so it equals the module's cyclomatic complexity minus one.

For a function f, its *own* nodes are its descendants that do not lie inside a nested def or class. Then:

- *length* = last line − def line +1: physical lines including the docstring, comments and blank lines, excluding decorators;
- *statements* = the number of own statement nodes (a nested def counts as one);
- *complexity* (McCabe, 1976) = 1+ the number of decision points among own nodes: each if/elif, for, while, except handler, conditional expression and assert adds 1; a boolean chain a and b and c with k operands adds $k - 1$; a comprehension adds 1 plus one per if filter; each case of a match adds 1;
- *parameters* = positional-only + positional + keyword-only parameters, plus one each for *args and **kwargs (self and cls included), the long parameter lists of Fowler (1999);
- *maximum nesting* (after Harrison & Magel, 1981) and nested loops, from one walk over the function body:

```
walk(stmts, d, in_loop):            # the function body has d = 0
    for s in stmts, skipping nested def/class:
        record depth d
        if s is a loop and in_loop: nested_loops += 1
        inner = in_loop or (s is a loop)
        if s is `if`:
            walk(s.body, d + 1, inner)
            # an elif chain stays at the same depth
```

```
            if s.orelse is an `if`: walk(s.orelse, d, inner)
            else:                   walk(s.orelse, d + 1, inner)
        else:
            for B in (s.body, s.orelse, s.finalbody,
                      every except-handler body):
                walk(B, d + 1, inner)
max_nesting(f) = largest recorded depth
```

Loop, conditional, `with` and `try` bodies each add one level, and statements inside `match` cases are not visited. At file level, `nested_loops` sums over functions and `max_nesting` is the largest function value, so module-level code is not measured for nesting.

Halstead measures (Halstead, 1977). One pass over a Python file's parse tree counts operators and operands. *Operators*: each arithmetic, bitwise, unary and comparison operator (keyed by type); a boolean chain contributes $\max(1, k-1)$ occurrences of its operator; augmented assignments (keyed by type, e.g. +=); assignment = (plain, annotated and walrus); call `()`, subscript `[]` and attribute access `.`; the keywords `if`, `for`, `while`, `with`, `try`, `except`, `return`, `raise`, `assert`, `import`, `lambda`, `global`, `nonlocal`, `del`, `break`, `continue`, `pass`, `await`, `yield`, `yield from`, the conditional expression, `def` and `class`; and, per comprehension, one `for` plus one `if` per filter. *Operands*: identifiers, parameter names, attribute names, defined function and class names, and constants (distinguished by type and value). Container literals, keyword-argument names, slices and starred expressions are not counted. With n_1, n_2 the distinct and N_1, N_2 the total operators and operands, the program length N, volume V, difficulty D, effort E and delivered bugs B are

$$N = N_1{+}N_2, \quad V = N\log_2(n_1{+}n_2), \quad D = \tfrac{n_1}{2}\cdot\tfrac{N_2}{n_2}\ (0 \text{ if } n_2 = 0), \quad E = D{\cdot}V, \quad B = V/3000.$$

`halstead_length`, `halstead_volume`, `halstead_effort` and `halstead_bugs` are sums of N, V, E and B over the lens's Python files. `halstead_difficulty` is the unweighted mean of per-file D, and files without operands (for example an empty `__init__.py`) enter it as $D = 0$.

Maintainability index. Per Python file, the maintainability index MI is

$$\mathrm{MI} = \min\Big(100,\ \max\Big(0,\ \tfrac{100}{171}\big(171 - 5.2\ln V - 0.23\,CC - 16.2\ln LOC + 50\sin\sqrt{2.4\,CM}\big)\Big)\Big),$$

with V the file's Halstead volume, its cyclomatic complexity $CC =$ `branches` $+1$, its lines of code $LOC =$ `code_lines` and its comment density $CM =$ `comment_ratio` $\in [0, 1]$. MI is undefined, and excluded, when $V = 0$ or $LOC = 0$. This is the four-term index of Oman & Hagemeister (1992); Coleman et al. (1994) rescaled to 0–100 as in Visual Studio (Microsoft, 2025), which itself drops the comment term. Keeping that term means a comment-dense file can gain up to 29 points. `avg_maintainability_index` and `min_maintainability_index` are the mean and minimum over the lens's files where MI is defined.

Coupling and cohesion. Python is dynamically typed, so the object-oriented metrics are static approximations resolved by name. ***CBO***, coupling between objects, from the Chidamber–Kemerer suite (Chidamber & Kemerer, 1994): for a class C, let $\mathit{refs}(C)$ be the names of its base classes plus every identifier and attribute name that appears anywhere in its body, and K the set of class names defined in any Python file of the repository. Then $\mathrm{CBO}(C) = |\mathit{refs}(C) \cap K \setminus \{C\}|$. Because matching uses bare names, classes from the standard library and third-party packages are ignored and same-named project classes are conflated. ***LCOM4***, the fourth formulation of lack of cohesion in methods (Hitz & Montazeri, 1995):

```
M     <- methods defined directly in the class body
A(m)  <- attribute names used on m's first parameter (self/cls)
link m1, m2 if A(m1) & A(m2) != {}     # they share a field
            or name(m2) in A(m1)       # m1 calls self.m2()
LCOM4 <- connected components of the method graph  # 1 = cohesive
```

Classes without methods (LCOM4 $= 0$) are excluded from `avg_lcom4` and `max_lcom4`. `non_cohesive_classes` counts classes with LCOM4 > 1, and `classes_measured` is the number of classes in the lens's Python files. *Fan-in and fan-out* (Henry & Kafura, 1981) use the

project's import graph at module level. A module's name comes from its repository-relative path (`a/b/c.py` → `a.b.c`; `__init__.py` names its package). Each import target is resolved, with relative imports taken against the importer's package, to the longest project module that equals it or is a dotted prefix of it. Standard-library, third-party and self imports are dropped. fan_out(m) counts the distinct project modules m imports, and fan_in(m) counts the distinct project modules that import m. The graph spans the whole repository, and the lens metrics are the mean and maximum over the lens's Python modules, empty modules included as 0. Because names derive from paths, absolute imports of a package kept under a `src/` directory do not resolve and are not counted.

Duplication. Duplication is detected on normalised lines, a language-independent approach (Ducasse et al., 1999; Roy et al., 2009):

```
for each analysed file:
    L <- its code lines (comments removed), each normalised:
         string literal -> S, digit run -> 0, whitespace collapsed;
         drop lines under 4 characters or with no letter or digit
    for each window w of 6 consecutive lines of L:
        record hash(w) at w's first line
cluster <- a hash recorded at >= 2 places in the repository
dup(f)  <- union, over f's occurrences of cluster hashes, of the
           6 physical lines starting at the occurrence's first line
```

Identifiers are not normalised, so only copies that are identical up to literals, spacing, blank lines and comments are found, including copies within one file. For a lens, `duplicated_code_lines` sums duplicated lines over its files (the other copy may lie outside the lens), `duplication_ratio` = `duplicated_code_lines` / `code_lines`, and `clone_clusters` counts the cluster hashes with at least one occurrence in the lens. A duplicated block of $k \geq 6$ normalised lines yields $k - 5$ overlapping windows, and so counts as $k - 5$ clusters.

Aggregation within a lens. Table 4 gives, for each metric, the unit it is measured on and how a lens combines the units in its file set. Stored means are rounded: function-level means and MI to one decimal, Halstead volume and effort to one decimal, and Halstead difficulty and bugs, CBO, LCOM4 and fan means to two decimals. When a lens contains no Python function, class or module, the corresponding means and maxima are missing and are treated as missing by the standardisation of Eq. 1. Sums, Halstead difficulty, and the function-length and nesting summaries default to 0.

Cognitive complexity (Campbell, 2018). One walk per function applies the three basic criteria of the original specification, which it labels B1 to B3. B1 withholds any increment for shorthand that lets several statements read as one, on the grounds that it does not burden the reader. B2 charges one increment for each break in the linear flow of the code, such as a conditional or a loop. B3 charges a further increment equal to the current nesting level whenever one such structure sits inside another, so the same construct costs more the deeper it is buried. In Python, `if`, a conditional expression, `for`, `while`, each `except` handler and `match` take $1 + n$ at nesting level n and raise the level for what they contain; `elif`, `else` and a loop's `else` take 1 with no nesting penalty; every maximal run of one boolean operator takes 1 (`a and b and c` once, `a and b or c` twice); a call to the enclosing function takes 1 for recursion; a nested `def` or `lambda` raises the nesting level without an increment; `with` and `try` bodies do neither, as they do not break the flow. `avg_cognitive_complexity` and `max_cognitive_complexity` are the mean and maximum over the lens's functions. The implementation reproduces the worked examples of Campbell (2018), including `sumOfPrimes` (6 in Python, where the labelled `continue` of the Java original is not available) and an `if`/`elif`/`else` chain (3).

Inheritance and response (Chidamber & Kemerer, 1994). These are the suite's four remaining metrics, beside CBO and the LCOM4 cohesion variant above. `avg_wmc` and `max_wmc`, weighted methods per class, are the sum of the cyclomatic complexities of a class's own methods, the weighting the suite leaves open and tools conventionally fill with complexity. `avg_dit` and `max_dit`, depth of the inheritance tree, are the longest path from a class to the root of its inheritance tree, counting only bases that resolve to a class defined in the repository, so a class with no project-defined base has depth 0; a cycle terminates the walk. `avg_noc` and `max_noc`, number of children, count immediate subclasses inside the repository. `avg_rfc` and `max_rfc`, response for a class, are the size of the

Table 4: **Unit of measurement and lens aggregation** for the 47 metrics listed. The remaining metrics aggregate as the paragraphs that define them state, by mean and maximum over functions or classes, or as a share over the lens.

Metric(s)	Measured per	Lens value
`files`, `python_files`, `test_files`	file	number of analysed, Python-parsed and test-path files
`physical_lines`, `code_lines`, `comment_lines`, `blank_lines`, `word_count`, `code_chars`	file	sum
`functions`, `classes`, `imports`, `branches`, `loops`, `nested_loops`, `logical_lines`	file	sum
`comment_ratio`	lens	comment / (comment + code) lines
`max_nesting`	function	maximum
`avg_max_nesting`	function	mean of per-function maximum nesting
`avg_function_length`, `median_function_length`, `max_function_length`	function	mean, median, maximum physical length
`avg_function_statements`, `max_function_statements`, `avg_complexity`, `max_complexity`, `avg_params`	Python function	mean or maximum
`halstead_length`, `halstead_volume`, `halstead_effort`, `halstead_bugs`	Python file	sum
`halstead_difficulty`	Python file	mean (files without operands count as 0)
`avg_maintainability_index`, `min_maintainability_index`	Python file with $V, LOC > 0$	mean, minimum
`classes_measured`	class	count
`avg_cbo`, `max_cbo`	class	mean, maximum
`avg_lcom4`, `max_lcom4`, `non_cohesive_classes`	class with ≥ 1 method	mean, maximum, count with LCOM4 > 1
`avg_fan_in`, `max_fan_in`, `avg_fan_out`, `max_fan_out`	Python module	mean, maximum
`duplicated_code_lines`	file	sum
`duplication_ratio`	lens	`duplicated_code_lines` / `code_lines`
`clone_clusters`	duplicated window	number with an occurrence in the lens

response set, the class's own methods plus the distinct method names its body invokes, in the usual depth-one reading. All four share the by-name resolution of CBO, and its limits.

Documentation and type annotations. `docstring_coverage` is the share of definitions (the module, each class and each function) whose first statement is a string literal, the convention of Goodger & van Rossum (2001) and the definition dedicated coverage tools use. Because the lexical pass counts a docstring as code rather than as a comment, this is the only measure of documentation in the set. `type_annotation_coverage` is the share of annotatable slots that carry an annotation (van Rossum et al., 2014), counting each function's parameters and its return, and excluding `self` and `cls`.

Readability features (Buse & Weimer, 2010). Readability models are fitted on surface features of the text, and we take those features directly, over a file's code lines (comment-only and blank lines excluded). `avg_line_length` and `max_line_length` are in characters; `avg_indentation` is the mean number of leading whitespace characters; `blank_line_ratio` is blank lines over all physical lines; `avg_identifier_length` and `single_char_identifier_ratio` are computed over identifier-like tokens. `naming_convention_ratio` is the share of functions named in `snake_case` and classes in `CapWords`, as van Rossum et al. (2001) prescribes, allowing leading and trailing underscores.

Package dependencies (Martin, 1994). On the module import graph already built for fan-in and fan-out, a module's efferent coupling C_e is the number of project modules it imports and its afferent coupling C_a the number that import it. `avg_instability` is the mean instability $I = C_e/(C_a + C_e)$ over the lens's modules with at least one dependency. `abstractness` is the share of the lens's classes that inherit from `ABC` or `Protocol` or declare an `abstractmethod`. `distance_main_sequence` is $|A + I - 1|$, for the lens's abstractness A and mean instability I, the distance from the line the same paper argues a well-placed package sits on. `cyclic_module_ratio` is the share of the lens's modules that lie in a cycle of the import graph, found as the strongly connected components with more than one member or a self-edge.

Exception handling. `bare_except_ratio` is the share of handlers written as a bare `except:`, which van Rossum et al. (2001) advises against, and `broad_except_ratio` the share that name `Exception` or `BaseException`. Both are over the lens's handlers.

Non-Python files. Other code files are measured lexically, using each language's comment and string syntax. For brace languages (JavaScript, TypeScript, C, Java, Go, ...) functions, classes, loops, nested loops and control nesting are read from the headers of `{ ... }` blocks, and function lengths from block extents. For other languages functions and classes are matched by regular expression and nesting is read from indentation. Branches are counted from decision keywords (`if`, `elif`, `for`, `foreach`, `while`, `case`, `when`, `catch`, `except`) and the operators `&&`, `||` and `?`. Such files contribute to the file, line, word and character counts, to the structural counts, to function length and to duplication, but not to logical lines, statements, complexity, parameters, Halstead, MI, CBO, LCOM4 or fan-in/fan-out.

Worked example. Running the analyser on the two files below, together with an empty `shapes/__init__.py`, gives the values in Table 5. `shapes/geometry.py`:

```
"""Geometry helpers."""
import math


class Circle:
    def __init__(self, r):
        self.r = r

    def area(self):
        return math.pi * self.r ** 2

    def label(self, prefix="c"):
        # no self.r here: a second, unconnected component
        return prefix.upper()


def total_area(shapes, min_r=0):
    total = 0
    for s in shapes:
        if s.r > min_r and s.r < 10:
            total += s.area()
    return total
```

`shapes/report.py`:

```
from .geometry import Circle, total_area


class Report:
    def __init__(self, radii):
        self.circles = [Circle(r) for r in radii if r > 0]

    def text(self):
        area = total_area(self.circles)
        return f"{len(self.circles)} circles, area {area:.1f}"
```

Table 5: **The worked example as measured by the analyser.** *Top*: per function. *Bottom*: per file, and two lens aggregates.

Function	Length	Statements	Complexity	Max nesting	Params
`Circle.__init__`, `Circle.area`	2	1	1	0	2, 1
`Circle.label`	3	1	1	0	2
`total_area`	6	5	4	2	2
`Report.__init__`	2	1	3	0	2
`Report.text`	3	2	1	0	1

Unit	Values
`geometry.py`	22 physical lines = 15 code (including the docstring) + 1 comment + 6 blank; $n_1 = 15$, $n_2 = 22$, $N_1 = 28$, $N_2 = 39$, so $V = 67 \log_2 37 = 349.0$ and $D = \frac{15}{2} \cdot \frac{39}{22} = 13.30$; MI = 67.0; fan-in 1
`report.py`	$V = 183.4$, $D = 8.57$, MI = 65.3; fan-out 1 (the relative import resolves to `shapes.geometry`)
`Circle`	LCOM4 = 2 (`label` shares no attribute with the other methods), CBO = 0
`Report`	LCOM4 = 1, CBO = 1 (it references `Circle`)
lens {`geometry.py`}	`halstead_difficulty` = 13.30, `avg_max_nesting` = $(2 + 0 + 0 + 0)/4 = 0.5$, `non_cohesive_classes` = 1
whole package	`halstead_difficulty` = $(0 + 13.30 + 8.57)/3 = 7.29$ (the empty `__init__.py` enters as $D = 0$), `avg_max_nesting` = $2/6 = 0.3$, `avg_complexity` = $11/6 = 1.8$

In `total_area`, complexity is 1 (base) +1 (`for`) +1 (`if`) +1 (the two-operand `and`) = 4, and the augmented assignment sits two levels deep, inside `for` and `if`. In `Report.__init__`, the comprehension with one filter adds 2, giving 3.

C LLM JUDGE PROMPT

Each task was judged in a self-contained folder. The folder held the task description given to the agents (`INSTRUCTION.md`), the base repository checked out at the task's base commit (`repo/`), one unified diff per candidate (`candidates/A.patch`, `B.patch`, ...) and the prompt below (`PROMPT.md`). The candidates were one passing completion per judged model, plus the gold patch. The judged models are 9 of the best performing on DeepSWE, each at its best performing reasoning effort, and each model's candidate is the passing sample whose sample identifier is the median of that model's, an arbitrary but reproducible choice. They were shuffled with a fixed seed and relabelled with letters. The mapping from labels to models was stored outside the folder and used only after judging. The judge, Claude Fable 5.1, could read only files inside the folder and could not search the web. We use the `ranking` and `scores` fields of its output. The prompt is reproduced below exactly as given, except that line breaks are reflowed to fit the page and em-dashes are typed as `-`.

```
# Code-quality review -- rank and score the implementations

You are a senior software engineer doing a code review. Several
developers each implemented the **same task** in the **same
repository**, independently. They are all functionally correct (they
pass the same hidden tests), so **do not judge correctness** -- judge
**code quality only**.

## What the patches implement
Read `INSTRUCTION.md` in this folder -- it is the task the developers
were given.

## The repository (its state BEFORE any patch)
`./repo/` is the base repository. Open any file there for context --
the existing conventions, the surrounding code a patch modifies, etc.

## The candidates
`./candidates/` has one file per developer: `A.patch`, `B.patch`, ...
-- each a unified diff of that developer's change. They are in
**random order and anonymised**; make no assumption from the labels.
Read each patch; consult `./repo/` for the context of what it changes.

## What to judge (code quality only)
For every candidate, weigh:
- **maintainability** -- clear structure, low complexity, no needless
  duplication, safe to change later;
- **extensibility** -- sensible abstractions that would be easy to
  extend or adapt, without over-engineering;
- **understandability** -- readable, well-named, appropriately
  commented, easy to follow.

## Output -- write `verdict.json` in this folder, EXACTLY this shape
   (no markdown fences):
{
  "ranking": ["<label>", "<label>", ...],
        // ALL candidate labels, best quality first
  "scores": { "<label>": <int 0-100>, ... },
        // one 0-100 score per candidate (higher = better)
  "per_candidate": { "<label>": "<1-3 sentence justification>", ... },
  "remarks": "<task-level remarks after ranking/scoring: notable
        patterns, near-ties, common pitfalls, or '' if none>"
}
Scores are ABSOLUTE quality on 0-100 (ties allowed); `ranking` must be
consistent with `scores` (higher score = earlier in ranking). Include
every candidate present in ./candidates/.

Read only files inside this task folder. Write only `verdict.json`. Do
not use web search.
```

D STATISTICAL DEFINITIONS

Within-task blocking. Tasks differ in difficulty, repository size and metric scale, so any statistic pooled across tasks would largely measure the task. Every figure is therefore computed *within* a task and then averaged across tasks, a repeated-measures design in which the task effect cancels.

Within-task robust standardisation. Write x_{ij} for the value of metric j on completion i of task t. Its standardised value is

$$z_{ij} \;=\; d_j \cdot \mathrm{clip}\!\left(\frac{x_{ij} - \mathrm{med}_t(x_{\cdot j})}{1.4826\ \mathrm{MAD}_t(x_{\cdot j})},\, -4,\, 4\right), \tag{1}$$

where med_t is the median over task t's candidates, MAD_t is their *median absolute deviation*, the median of $|x_{\cdot j} - \mathrm{med}_t(x_{\cdot j})|$, $\mathrm{clip}(u, -4, 4)$ truncates u to that interval, and $d_j \in \{+1, -1\}$ orients directional metrics so higher always means better. The factor 1.4826 makes the scale comparable to a standard deviation under normality (Rousseeuw & Croux, 1993), and clipping prevents one outlier from dominating a fit. When the MAD is zero, common for small-integer metrics, we fall back to the standard deviation. Context metrics, having no better direction, enter as measured ($d_j = +1$), so larger patches score higher on them.

Rank correlation and concordance. Both measures compare two orderings of one task's n candidates, the signal's and the judge's. Spearman's ρ (Spearman, 1904) is Pearson correlation on ranks. Kendall's τ_b (Kendall, 1938) works on the $n_0 = n(n-1)/2$ pairs of candidates, counting those the two orderings rank the same way (n_c, concordant) and those they rank oppositely (n_d, discordant),

$$\tau_b \;=\; \frac{n_c - n_d}{\sqrt{(n_0 - n_1)(n_0 - n_2)}},$$

where the tie corrections n_1 and n_2 each sum $m(m-1)/2$ over the groups of m equal values, in the signal and in the judge's scores respectively. By Cohen's convention $|\cdot| \approx 0.1/0.3/0.5$ is small/moderate/large (Cohen, 1988). The headline measure is the probability that a predictor orders a random within-task pair the way the judge did,

$$\text{concordance} = \frac{\tau + 1}{2} \in [0,\, 1], \tag{2}$$

exact for untied data and a close approximation with ties. Chance is 0.5, a signal that orders every pair as the judge does reaches 1, and a signal that runs against the judge falls below 0.5. A task on which the predictor is constant orders nothing and is scored at chance ($\rho = \tau = 0$) rather than dropped.

AUC. For the binary correctness target the counterpart is the area under the receiver operating characteristic curve, AUC (Hanley & McNeil, 1982), equivalently the Mann–Whitney statistic (Mann & Whitney, 1947): the probability that a random passing completion scores above a random failing one from the same task, ties counting a half. It is concordance with the judge's ordering replaced by the pass/fail split, on the same 0.5–1 scale, which is what makes the two comparable, and it is invariant to any monotone rescaling.

Significance and multiplicity. Per-task statistics are averaged and tested with a one-sample t-test over the tasks on which the signal varies (the *Tasks* column of Table 12), at most the $T = 31$ scorable tasks, so tasks rather than completions set the power. That test uses all of those tasks, whereas the ρ we report is held out. Testing every metric–lens combination at $p < 0.05$ would return about eleven false positives by chance alone, so we instead control the **false-discovery rate** (FDR), the expected share of false positives among the combinations we call aligned, by Benjamini–Hochberg (Benjamini & Hochberg, 1995). With M the number of tests and $p_{(1)} \le \cdots \le p_{(M)}$ the sorted p-values, the k-th smallest is assigned $q_{(k)} = \min_{j \ge k} M p_{(j)}/j$, and we read $q < 0.05$ as significant.

That test is two-sided, so a small q says only that the metric orders candidates consistently with the judge. It does not say that the metric orders them the way the measurement literature expects. The sign of ρ says that. Eq. 1 has already flipped each directional metric so that higher means better by its literature direction, so a positive mean ρ means the judge agrees with that direction, and a negative

one means the judge systematically prefers the opposite. We therefore call a metric *aligned* when $q < 0.05$ and its mean ρ is positive, and say it *runs the other way* when $q < 0.05$ and its mean ρ is negative. The twelve that run the other way (§5.1) are mostly size-driven counts, on which the judge ranks the larger patch higher although the literature reads more code as worse.

Penalised regression. The linear index is a score $s = \beta_0 + \sum_j \beta_j z_j$ fitted by penalised least squares with the L1 penalty $\lambda\|\beta\|_1 = \lambda \sum_j |\beta_j|$ of the LASSO (Tibshirani, 1996), where $\lambda > 0$ sets how hard the weights are pulled towards zero and the intercept β_0 is never penalised. An L1 penalty drives weights exactly to zero, so the fit also selects features. A penalty is not optional: the 215 features are strongly collinear, with a median variance inflation factor, VIF, of 24.4 (a feature's VIF is $1/(1 - R_j^2)$ for the R_j^2 of regressing it on the others, the factor by which collinearity inflates its coefficient's variance), so unpenalised least squares is unidentifiable. The L1 fits use cyclic coordinate descent with soft-thresholding (Friedman et al., 2010).

Evaluation and tuning protocol. Every fitted signal in Table 1 follows the same five steps, so no method is given a freedom the others are denied.

1. *Outer split.* The 33 tasks are partitioned $80/20$, 26 training and 7 test. Whole tasks move together, so no repository is on both sides and the unit of resampling is the task.
2. *Inner tuning.* The training tasks are divided into 4 folds, again whole tasks, and the hyperparameter with the highest mean within-task Kendall τ over held-out folds is kept. The tuning criterion is the quantity we report, not a surrogate such as R^2.
3. *Refit.* That setting is refitted on all 26 training tasks.
4. *Test.* The fit scores the 7 test tasks once, and no judge label from a test task enters it. Two preprocessing steps use no labels and are computed once over all judged completions, the per-feature centring and scaling, which the tree ensembles are invariant to, and the filter that drops columns that are missing or constant too often.
5. *Repeat.* Steps 1–4 run over 300 random partitions, pooled. All approaches see identical partitions, so the rows of Table 1 are paired.

Single metrics are not fitted; each is oriented by its literature direction and scored on the same test tasks, so every effect size is held out and the rows are comparable. The grids searched in step 2 are LASSO $\lambda \in \{0.005, \ldots, 0.20\}$; XGBoost over (tree depth, boosting rounds) $\in \{(1, 300), (2, 200), (3, 120)\}$ crossed with a column subsample of $\{0.05, 0.5\}$; and Extra Trees over the share of features offered at each split, $\{0.03, 0.05, 0.10, 0.20\}$. Both ensembles are offered the same aggressive subsampling range, so neither gains a decorrelation advantage by construction.

The standard errors in Tables 1 and 12. Every row, fitted or not, yields one ρ and one τ for each (split, test task) pair, and the tables report their average. The 300 splits revisit the same $T \leq 31$ tasks, so those figures are repeated measurements of a few units rather than independent draws, and treating each as independent would understate the uncertainty several-fold. We therefore cluster by task. All of a task's values are averaged over the splits in which it was a test task, leaving one number per task, and the reported standard error is that of the task means, $s/\sqrt{T}$ for their sample standard deviation s.

From patch scores to model scores. Two standardisations run in parallel, both by Eq. 1 and differing only in the comparison group. A *patch* is standardised against the other candidates for its task. That is the unit the judge ranks, so it is what the alignment analysis measures and what every index is fitted on and applied to. A *model*'s number in Table 2 and Figure 1 is built in three steps instead. Its completions on a task are collapsed to one value, the median of the raw values for a single metric and the mean of the patch scores for a fitted index, which are already standardised and clipped per patch. Those per-task values are standardised across the 26 models by Eq. 1, now with the models as the comparison group and a task whose models have zero MAD contributing 0. The clipped z's are then averaged over the tasks the model has completions for. Gold is placed on the models' scale the same way, for an index from a refit that excludes the gold rows, and never enters the models' median or MAD. *Overall* averages a model's 216 feature scores with equal weight.

Task-held-out model scores. The leaderboard fits the aligned index on all judged completions, at the value of `max_features` the inner cross-validation chooses most often across the 300 splits,

and scores the unjudged rest. `max_features` is the Extra Trees hyper-parameter tuned in step 2 above, the share of the features the ensemble may consider when it chooses each split of a tree.

The models are already held out. The judge saw 9 of the 26 models, so the other 17 contribute no label to any fit and their placement is a prediction rather than a fitted value, which Appendix G tests directly with a second judge. Tasks are not held out in the same way. Every completion the index scores, judged or not, comes from the same 33 tasks, so a fit could in principle learn what the judge rewarded in one particular repository and then be credited for scoring that repository's other completions. We therefore cross-fit by task, leaving one out at a time. Each of the 33 fits excludes one task's judged completions, takes its feature scaling from the other tasks, and scores only that task, so no judge label from a task enters the scores of that task. The resulting ranking of the 26 models matches the leaderboard's at $\rho = +0.999$, and its agreement with the judge is unchanged ($+0.95$ over the 9 judged models against $+0.95$ for the leaderboard, and $+0.96$ with gold against $+0.96$). Gold's patches fall further under these fits, to -1.34, so the leaderboard's refit without the gold rows is the conservative choice.

Which gradient booster. On 60 of the same splits with fixed hyper-parameters, CatBoost, LightGBM and XGBoost all land within a point of each other, between 78.1% and 79.1%. We report XGBoost, the most widely used of the three, under a squared-error objective, which a pairwise ranking objective taking each task as a query group matched to within a rounding step.

E Metric Alignment with Correctness

The main study holds correctness constant and asks which metrics track *quality*. This supplementary study asks the complementary question. Across all completions that applied cleanly, passing *and* failing, can the same metrics tell the two apart? It serves two purposes. It tests whether the quality signal of §5.1 is a covert correctness signal, and it exposes the size confound in its purest form.

Design. We use all 1,173 cleanly-applied completions that DeepSWE scores from the 9 judged models at their best performing efforts (809 pass / 364 fail; 69.0% pass rate; median 4 completions per task–model cell). Gold is excluded as it passes by construction. The target is binary, so the measure is the within-task AUC of Appendix D, averaged over tasks, tested against 0.5 with a t-test across tasks and FDR-controlled across the 216 metric–lens features.

Table 6: **Best-aligned metrics for correctness** (within-task AUC, chance $= 0.5$). Four of the five are *size/context* measures with no better direction, and none is significant (q above 0.99 for every row after FDR correction).

Metric	Lens	Dir.	Within-task AUC
`imports`	focus	·	**0.577**
`code_lines`	focus	·	0.561
`physical_lines`	focus	·	0.555
`code_chars`	focus	·	0.547
`avg_indentation`	union	↓	0.547

Findings. None of the 216 metric–lens features separates passing from failing completions better than chance at FDR $q < 0.05$. The largest within-task AUC is 0.577, and every q is above 0.99. What little lean there is, is size, not cleanliness. Four of the five rows of Table 6 are size/context counts, and among metrics with a genuine quality direction the strongest (`avg_indentation`, `avg_cbo`, `max_cognitive_complexity`, `max_complexity`, all ↓) reach only 0.53–0.55, in the unexpected direction, with messier code passing slightly more often. That is the pattern expected if a failing completion differs in how much of the task it attempted rather than in how cleanly it did so, but none of it is distinguishable from chance.

Why this matters for the main result. The two studies reach opposite results. Duplication, commenting, nesting and function size align with judged quality, whereas no metric reliably predicts correctness. Their strengths differ accordingly (0.80 held-out concordance for the aligned index

against a best AUC of 0.58 that is not significant). Static quality metrics are therefore substantially better at predicting *how good code looks to an expert* than *whether it works*, which is both what one would hope and a necessary sanity check on §5.1.

F JUDGE AGREEMENT

Table 7 compares the paper's judge, Claude Fable 5.1, with Claude Opus 5 run on the same blinded candidates. Table 8 compares Claude Fable 5.1 with the preliminary human rankings (§5.3). The raters were not given the judge's prompt. They ranked the candidates on whichever aspects of code quality they themselves preferred. Self-preference is a documented failure mode of LLM judges (§2), and both judges here are Anthropic models, so it is worth stating that neither favours the model from its own family: each ranks `claude-opus-5` and `claude-fable-5` in the same positions the other does, and the two orderings agree exactly.

Table 9 measures the metrics against preliminary human rankings. The table stops at the single metrics and *Overall*. If a combined index were to be trained, the eight tasks carrying 64 patches leave a fit with more variance than the differences it would be asked to resolve.

Seventy-eight of the 147 directional combinations point the way the raters do. The closest is total indentation at 0.648 ± 0.056, level with the judge's own 0.652 ± 0.050 against these raters, and duplication, nesting and line length follow within one standard error of it. *Overall* reaches 0.589 ± 0.036. Cognitive complexity, the strongest metric against the judge (Table 1), is 18th here at 0.583 ± 0.050. Across all 147 combinations the two targets rank the metrics similarly (Spearman $\rho = +0.46$ between human-side and judge-side concordance), so they largely favour the same metrics and disagree on which one leads. With one rater per task and standard errors near 0.05, however, the ordering inside the table is not resolved, and we draw no conclusion about individual metrics from it.

Table 7: **Claude Fable 5.1 against Claude Opus 5** on the same blinded candidates (33 tasks, 290 completions, one passing completion per model for 9 of the best performing models, each at its best performing reasoning effort). *Score* is the mean 0–100 score, *Norm. rank* the mean normalised rank (0 = best in task) and *Pos.* the position by normalised rank. The two orderings are identical (Spearman $\rho = +1.00$), and the judges order 88.5% of the 1,204 within-task pairs the same way.

Model	Claude Fable 5.1			Claude Opus 5		
	Score	Norm. rank ↓	Pos.	Score	Norm. rank ↓	Pos.
`glm-5-3`	77.3	0.18	1	75.8	0.17	1
`claude-opus-5`	76.4	0.23	2	75.3	0.20	2
`claude-fable-5`	75.0	0.25	3	73.5	0.25	3
`kimi-k3`	74.8	0.27	4	72.7	0.27	4
`gpt-5-6-sol`	68.4	0.41	5	65.7	0.45	5
`grok-4-6`	65.8	0.50	6	61.4	0.52	6
`gpt-5-5`	58.8	0.64	7	57.1	0.64	7
`gpt-5-6-luna`	56.2	0.68	8	55.8	0.65	8
`gemini-3-7-flash`	47.1	0.83	9	40.6	0.88	9
gold (reference)	37.0	0.94	10	37.1	0.92	10

Table 8: **Claude Fable 5.1 against the preliminary human rankings** on the eight rated tasks (seven models and gold per task, 224 within-task pairs). The candidates are the completions the raters saw, one passing completion per model from an earlier sample of DeepSWE trials that pooled every reasoning effort, and the judge ranked the same eight per task blind. *Mean rank* is taken within each task's eight candidates (1 = best) and averaged over the eight tasks, *Pos.* is the position by mean rank (ties share a position) and *Score* is the judge's mean 0–100 score. Human scores are not used, since half the raters gave rankings only. The judge orders 65% of the raters' within-task pairs the same way (chance 50%). With one rater per task, the table supports no conclusion about individual models.

Model	Human raters		Claude Fable 5.1		
	Mean rank ↓	Pos.	Mean rank ↓	Pos.	Score
`claude-opus-5`	2.50	1	4.63	5	67.1
`claude-fable-5`	3.63	2	2.25	2	78.8
`claude-sonnet-5`	4.25	3	4.13	4	71.6
`gpt-5-6-sol`	4.25	3	4.63	5	65.1
`kimi-k3`	4.25	3	2.13	1	78.4
`grok-4-5`	5.13	6	6.25	7	58.5
`gpt-5-6-terra`	5.38	7	4.00	3	69.6
gold (reference)	6.63	8	8.00	8	34.5

Table 9: **Alignment with the preliminary human rankings** on the eight rated tasks (224 within-task ordered pairs), in the format of Table 1. The five single metrics closest to the raters, then *Overall*, the unweighted average of all 216 features, with the paper's judge on the same pairs for reference. *Aligned* ρ and *Concordance* are averaged over tasks, $\pm$ one task-clustered standard error. Concordance 0.5 is chance. Combined indices are omitted, since eight tasks are too few to fit one.

Signal	Lens / method	Aligned ρ	Concordance
`avg_indentation`	total	$+0.386 \pm 0.133$	0.648 ± 0.056
`duplication_ratio`	total	$+0.295 \pm 0.119$	0.634 ± 0.040
`avg_max_nesting`	focus	$+0.328 \pm 0.103$	0.629 ± 0.043
`duplication_ratio`	union	$+0.244 \pm 0.125$	0.621 ± 0.043
`max_line_length`	focus	$+0.287 \pm 0.083$	0.614 ± 0.033
Overall	all 216 features	$+0.268 \pm 0.103$	0.589 ± 0.036
Claude Fable 5.1, the paper's judge	—	$+0.372 \pm 0.118$	0.652 ± 0.050

G Sanity Check on the Unjudged Models

The index ranks all 26 models, but the judge saw only nine of them (§4.3), so the models at the bottom of the leaderboard carry no judge data of their own. We therefore ran one more blind round, under the protocol of §4.3 and with a different judge, Claude Opus 5.

The roster is the best model of each family on DeepSWE's leaderboard that released patches, at its best performing reasoning effort, plus the gold patch. The 10 tasks are those on which all nine models have a passing completion, five of them from the human study (§5.3). Every completion shared with the round of §4.3 is the one that round judged. Three of the nine models were never judged before, so their placement is an out-of-sample prediction of the index.

The judge's ranking of the nine models is almost exactly the index's (Spearman $\rho = +0.98$), and $+0.97$ against *Overall*, while against the pass rate it is only $+0.48$. The three models the judge had never seen land where the index placed them. `muse-spark-1-2`, the lowest-scoring model of the leaderboard, is last here too, at a mean score of 30.3. It is ranked worst in 9 of the 10 tasks and below the gold patch in 9. Its position is therefore judged, not extrapolated. Gold again sits near the bottom (mean normalised rank 0.89), above only `muse-spark-1-2`.

Table 10: **The best model of each family, judged blind by Claude Opus 5** on 10 tasks with 10 candidates each. *Judge score* is the mean 0–100 score and *Norm. rank* the mean normalised rank (0 = best in task). The *Aligned index* and *Pass rate* columns are those of Table 11. ‡Models the judge had never seen.

Model	Judge score	Norm. rank ↓	Aligned index	Pass rate (%)
`glm-5-3`	82.3	0.12	+1.07	66.4
`claude-opus-5`	82.9	0.14	+1.09	74.6
`kimi-k3`	81.0	0.22	+0.83	67.9
`qwen3-8-max`‡	73.6	0.39	+0.50	56.1
`grok-4-6`	72.1	0.43	+0.35	67.4
`gpt-5-6-sol`	71.7	0.47	+0.12	75.4
`deepseek-v4-pro`‡	65.5	0.60	−0.41	58.3
`gemini-3-7-flash`	58.3	0.74	−0.75	60.6
gold (reference)	**49.3**	**0.89**	**−0.86**	—
`muse-spark-1-2`‡	30.3	0.99	−1.99	50.0

H FULL TABLES

Table 11: **All 26 models and the gold reference, ranked by the SWE-Quality Aligned Index.** Each model is evaluated at its best performing reasoning effort (*Effort*, — for a model run without an effort setting). Columns are as in Table 2. The aligned index is scored only on completions it was not fitted on (†gold by a refit without its rows). Only 9 of the best performing models were judged, and the others have no LLM-judge values (—).

Model	Effort	Quality (z)		LLM judge (Claude Fable 5.1)		Correctness
		Aligned index	Cognitive	Score	Norm. rank ↓	Pass rate (%)
`claude-opus-5`	max	+1.09	+0.96	76.4	0.23	74.6
`glm-5-3`	max	+1.07	+0.82	77.3	0.18	66.4
`claude-sonnet-5`	max	+1.04	+1.43	—	—	59.1
`glm-5-3-flash`	max	+1.04	+0.77	—	—	71.5
`kimi-k3`	max	+0.83	+0.70	74.8	0.27	67.9
`claude-opus-4-8`	max	+0.77	+0.72	—	—	63.4
`claude-fable-5`	xhigh	+0.57	+0.57	75.0	0.25	73.5
`qwen3-8-max`	xhigh	+0.50	+0.49	—	—	56.1
`grok-4-6`	medium	+0.35	+0.19	65.8	0.50	67.4
`glm-5-2`	max	+0.27	+0.36	—	—	40.8
`claude-sonnet-4-6`	high	+0.21	+0.49	—	—	27.3
`gpt-5-6-terra`	max	+0.15	−0.02	—	—	68.2
`gpt-5-6-sol`	max	+0.12	+0.03	68.4	0.41	75.4
`deepseek-v4-flash`	max	−0.16	−0.01	—	—	47.7
`grok-4-5`	high	−0.18	−0.26	—	—	59.8
`gpt-5-4`	xhigh	−0.25	+0.04	—	—	44.7
`gpt-5-5`	xhigh	−0.28	+0.01	58.8	0.64	65.9
`deepseek-v4-pro`	max	−0.41	−0.54	—	—	58.3
`gpt-5-6-luna`	max	−0.53	−0.75	56.2	0.68	64.4
`kimi-k2-7-code`	—	−0.60	−0.11	—	—	24.2
`gemini-3-7-flash`	medium	−0.75	−1.07	47.1	0.83	60.6
gold (reference)	—	**−0.86**†	**−1.04**	**37.0**	**0.94**	**100.0**
`gemini-3-6-flash`	high	−0.95	−1.21	—	—	40.2
`gemini-3-5-flash`	high	−1.28	−1.55	—	—	28.8
`gemini-3-1-pro-preview`	high	−1.66	−1.10	—	—	8.3
`muse-spark-1-1`	xhigh	−1.90	−3.08	—	—	51.5
`muse-spark-1-2`	xhigh	−1.99	−3.39	—	—	50.0

Metric alignment in full. Table 12 extends Table 1 to all 147 directional metric–lens combinations. The 42 that are significantly aligned at FDR $q < 0.05$ have held-out effect sizes mostly $\rho \approx 0.21$–0.32

and at most 0.65, small to large by Cohen's convention (Cohen, 1988). The leading families are cognitive complexity, readability, documentation and duplication, then the nesting, parameter-list and function-size measures the classical literature identifies.

Table 12: **Full metric alignment**, the complete version of Table 1 with all 147 directional metric–lens combinations, ranked by held-out aligned Spearman ρ against the LLM judge, Claude Fable 5.1. As in Table 1, each metric is oriented by its literature direction without fitting and scored only on the test tasks of the repeated $80/20$ task splits, with a standard error clustered by task. *Concordance* is the share of those tasks' within-task pairs ordered like the judge (chance $= 0.5$), and a task on which the metric is constant counts as chance. *Tasks* is how many of the 31 scorable tasks the metric varies on. * marks FDR $q < 0.05$ in the task-level t-test over the tasks on which the metric varies (Appendix D). A negative ρ points against the literature direction. Of the 12 significant negative rows, eight are counts that grow with the size of the change (Halstead effort, Halstead bugs, branches and non-cohesive classes). The other four are avg line length on the whole repository, median function length on the focus lens, single char identifier ratio on the union lens and the maintainability index on the whole repository.

Metric	Lens	Dir.	Aligned ρ	Tasks	Concordance
avg_cognitive_complexity	union	↓	$+0.651 \pm 0.047^*$	31/31	0.766 ± 0.020
avg_cognitive_complexity	total	↓	$+0.616 \pm 0.044^*$	31/31	0.748 ± 0.019
avg_indentation	union	↓	$+0.508 \pm 0.041^*$	31/31	0.696 ± 0.018
avg_indentation	total	↓	$+0.419 \pm 0.064^*$	31/31	0.667 ± 0.026
docstring_coverage	focus	↑	$+0.410 \pm 0.068^*$	31/31	0.665 ± 0.026
duplication_ratio	total	↓	$+0.374 \pm 0.071^*$	31/31	0.648 ± 0.029
comment_ratio	focus	↑	$+0.372 \pm 0.058^*$	31/31	0.634 ± 0.022
avg_max_nesting	union	↓	$+0.341 \pm 0.056^*$	24/31	0.641 ± 0.023
avg_params	union	↓	$+0.332 \pm 0.078^*$	27/31	0.640 ± 0.034
duplication_ratio	union	↓	$+0.327 \pm 0.068^*$	31/31	0.625 ± 0.027
avg_cognitive_complexity	focus	↓	$+0.319 \pm 0.071^*$	31/31	0.629 ± 0.028
avg_function_statements	union	↓	$+0.315 \pm 0.075^*$	31/31	0.625 ± 0.031
duplication_ratio	focus	↓	$+0.311 \pm 0.059^*$	31/31	0.626 ± 0.024
avg_function_length	union	↓	$+0.290 \pm 0.076^*$	31/31	0.612 ± 0.029
avg_indentation	focus	↓	$+0.284 \pm 0.077^*$	31/31	0.608 ± 0.031
branches	focus	↓	$+0.278 \pm 0.079^*$	31/31	0.615 ± 0.031
naming_convention_ratio	total	↑	$+0.268 \pm 0.063^*$	22/31	0.602 ± 0.026
halstead_difficulty	focus	↓	$+0.260 \pm 0.073^*$	31/31	0.602 ± 0.029
max_line_length	focus	↓	$+0.258 \pm 0.055^*$	18/31	0.609 ± 0.024
avg_complexity	union	↓	$+0.258 \pm 0.079^*$	30/31	0.604 ± 0.032
avg_function_statements	total	↓	$+0.254 \pm 0.058^*$	22/31	0.607 ± 0.024
avg_function_length	total	↓	$+0.253 \pm 0.060^*$	27/31	0.600 ± 0.024
avg_complexity	focus	↓	$+0.252 \pm 0.070^*$	30/31	0.611 ± 0.027
max_function_statements	focus	↓	$+0.239 \pm 0.063^*$	29/31	0.596 ± 0.025
halstead_effort	focus	↓	$+0.236 \pm 0.081^*$	31/31	0.603 ± 0.032
max_line_length	union	↓	$+0.229 \pm 0.054^*$	17/31	0.598 ± 0.023
docstring_coverage	total	↑	$+0.225 \pm 0.081^*$	31/31	0.587 ± 0.031
clone_clusters	focus	↓	$+0.223 \pm 0.057^*$	30/31	0.588 ± 0.025
max_complexity	union	↓	$+0.222 \pm 0.062^*$	23/31	0.592 ± 0.026
avg_function_statements	focus	↓	$+0.214 \pm 0.065^*$	30/31	0.587 ± 0.025
duplicated_code_lines	focus	↓	$+0.214 \pm 0.063^*$	31/31	0.583 ± 0.026
max_cognitive_complexity	focus	↓	$+0.209 \pm 0.071^*$	29/31	0.583 ± 0.029
max_cognitive_complexity	union	↓	$+0.202 \pm 0.063^*$	26/31	0.579 ± 0.026
blank_line_ratio	union	↑	$+0.192 \pm 0.082$	31/31	0.571 ± 0.032
max_cognitive_complexity	total	↓	$+0.184 \pm 0.054^*$	15/31	0.576 ± 0.023
avg_max_nesting	focus	↓	$+0.184 \pm 0.065^*$	27/31	0.576 ± 0.027
max_complexity	focus	↓	$+0.170 \pm 0.061^*$	26/31	0.569 ± 0.026
max_complexity	total	↓	$+0.165 \pm 0.058^*$	12/31	0.571 ± 0.025
avg_wmc	union	↓	$+0.163 \pm 0.084$	31/31	0.562 ± 0.033
halstead_bugs	focus	↓	$+0.162 \pm 0.084$	31/31	0.574 ± 0.033
max_function_statements	union	↓	$+0.161 \pm 0.061^*$	27/31	0.565 ± 0.025
loops	focus	↓	$+0.159 \pm 0.083$	31/31	0.566 ± 0.033
comment_ratio	union	↑	$+0.151 \pm 0.067$	31/31	0.547 ± 0.024
avg_params	total	↓	$+0.149 \pm 0.050^*$	11/31	0.563 ± 0.021

Table 12 continued

Metric	Lens	Dir.	Aligned ρ	Tasks	Concordance
avg_complexity	total	↓	$+0.140 \pm 0.060$	18/31	0.561 ± 0.026
max_function_statements	total	↓	$+0.139 \pm 0.059$	16/31	0.558 ± 0.024
avg_wmc	focus	↓	$+0.137 \pm 0.072$	31/31	0.549 ± 0.027
max_nesting	union	↓	$+0.135 \pm 0.046^{*}$	16/31	0.555 ± 0.019
max_line_length	total	↓	$+0.130 \pm 0.042^{*}$	9/31	0.557 ± 0.019
docstring_coverage	union	↑	$+0.123 \pm 0.077$	31/31	0.543 ± 0.031
distance_main_sequence	union	↓	$+0.123 \pm 0.064$	22/31	0.550 ± 0.026
cyclic_module_ratio	union	↓	$+0.121 \pm 0.047^{*}$	11/31	0.550 ± 0.020
avg_cbo	union	↓	$+0.115 \pm 0.079$	31/31	0.546 ± 0.031
max_nesting	focus	↓	$+0.112 \pm 0.051$	19/31	0.545 ± 0.022
blank_line_ratio	total	↑	$+0.108 \pm 0.089$	31/31	0.536 ± 0.035
blank_line_ratio	focus	↑	$+0.103 \pm 0.081$	31/31	0.536 ± 0.033
avg_wmc	total	↓	$+0.102 \pm 0.079$	31/31	0.542 ± 0.032
max_fan_out	focus	↓	$+0.098 \pm 0.052$	14/31	0.541 ± 0.023
max_wmc	focus	↓	$+0.097 \pm 0.071$	28/31	0.541 ± 0.027
nested_loops	focus	↓	$+0.097 \pm 0.066$	25/31	0.540 ± 0.026
max_wmc	union	↓	$+0.096 \pm 0.068$	29/31	0.541 ± 0.026
comment_ratio	total	↑	$+0.096 \pm 0.070$	31/31	0.524 ± 0.025
cyclic_module_ratio	total	↓	$+0.095 \pm 0.057$	14/31	0.539 ± 0.024
avg_maintainability_index	focus	↑	$+0.093 \pm 0.088$	29/31	0.538 ± 0.036
distance_main_sequence	total	↓	$+0.085 \pm 0.075$	26/31	0.542 ± 0.030
min_maintainability_index	focus	↑	$+0.084 \pm 0.060$	12/31	0.535 ± 0.023
naming_convention_ratio	union	↑	$+0.078 \pm 0.053$	16/31	0.530 ± 0.021
nested_loops	union	↓	$+0.078 \pm 0.067$	27/31	0.534 ± 0.027
max_nesting	total	↓	$+0.077 \pm 0.036$	8/31	0.532 ± 0.015
avg_params	focus	↓	$+0.077 \pm 0.053$	26/31	0.529 ± 0.023
type_annotation_coverage	focus	↑	$+0.072 \pm 0.054$	19/31	0.527 ± 0.020
nested_loops	total	↓	$+0.068 \pm 0.067$	27/31	0.530 ± 0.027
avg_cbo	focus	↓	$+0.063 \pm 0.063$	31/31	0.527 ± 0.026
avg_lcom4	union	↓	$+0.062 \pm 0.068$	29/31	0.517 ± 0.027
clone_clusters	total	↓	$+0.056 \pm 0.061$	31/31	0.521 ± 0.023
avg_lcom4	focus	↓	$+0.053 \pm 0.063$	24/31	0.521 ± 0.026
avg_cbo	total	↓	$+0.052 \pm 0.075$	27/31	0.519 ± 0.031
max_wmc	total	↓	$+0.051 \pm 0.059$	20/31	0.525 ± 0.022
naming_convention_ratio	focus	↑	$+0.051 \pm 0.028$	8/31	0.512 ± 0.009
avg_fan_out	union	↓	$+0.050 \pm 0.084$	30/31	0.520 ± 0.033
max_function_length	total	↓	$+0.045 \pm 0.041$	10/31	0.519 ± 0.016
max_rfc	union	↓	$+0.045 \pm 0.067$	29/31	0.518 ± 0.026
bare_except_ratio	total	↓	$+0.044 \pm 0.036$	9/31	0.517 ± 0.014
max_function_length	union	↓	$+0.041 \pm 0.073$	24/31	0.517 ± 0.029
avg_lcom4	total	↓	$+0.038 \pm 0.060$	28/31	0.511 ± 0.024
max_rfc	focus	↓	$+0.036 \pm 0.065$	28/31	0.517 ± 0.025
avg_max_nesting	total	↓	$+0.035 \pm 0.025$	6/31	0.515 ± 0.011
max_cbo	total	↓	$+0.035 \pm 0.057$	14/31	0.518 ± 0.024
avg_dit	total	↓	$+0.035 \pm 0.066$	26/31	0.516 ± 0.027
bare_except_ratio	union	↓	$+0.034 \pm 0.026$	2/31	0.513 ± 0.010
max_function_length	focus	↓	$+0.033 \pm 0.079$	26/31	0.514 ± 0.031
max_cbo	focus	↓	$+0.033 \pm 0.071$	25/31	0.518 ± 0.030
broad_except_ratio	union	↓	$+0.029 \pm 0.057$	24/31	0.505 ± 0.021
max_fan_out	union	↓	$+0.029 \pm 0.052$	13/31	0.511 ± 0.023
avg_rfc	union	↓	$+0.026 \pm 0.087$	31/31	0.511 ± 0.032
median_function_length	union	↓	$+0.026 \pm 0.081$	26/31	0.506 ± 0.033
min_maintainability_index	union	↑	$+0.022 \pm 0.046$	7/31	0.510 ± 0.018
min_maintainability_index	total	↑	$+0.021 \pm 0.024$	2/31	0.508 ± 0.009
clone_clusters	union	↓	$+0.019 \pm 0.064$	31/31	0.507 ± 0.025
bare_except_ratio	focus	↓	$+0.015 \pm 0.017$	1/31	0.505 ± 0.005
avg_dit	union	↓	$+0.013 \pm 0.066$	24/31	0.512 ± 0.026
broad_except_ratio	total	↓	$+0.008 \pm 0.062$	26/31	0.493 ± 0.023
broad_except_ratio	focus	↓	$+0.008 \pm 0.050$	22/31	0.498 ± 0.019
halstead_difficulty	union	↓	$+0.006 \pm 0.083$	31/31	0.502 ± 0.032
max_lcom4	focus	↓	$+0.004 \pm 0.034$	7/31	0.503 ± 0.014
max_cbo	union	↓	$+0.003 \pm 0.067$	21/31	0.505 ± 0.028

Table 12 continued

Metric	Lens	Dir.	Aligned ρ	Tasks	Concordance
max_rfc	total	↓	$+0.003 \pm 0.062$	22/31	0.502 ± 0.024
max_dit	total	↓	-0.004 ± 0.015	3/31	0.498 ± 0.006
avg_rfc	focus	↓	-0.004 ± 0.067	31/31	0.504 ± 0.024
median_function_length	total	↓	-0.006 ± 0.045	9/31	0.496 ± 0.019
single_char_identifier_ratio	focus	↓	-0.016 ± 0.079	31/31	0.482 ± 0.031
max_dit	focus	↓	-0.016 ± 0.024	5/31	0.494 ± 0.009
cyclic_module_ratio	focus	↓	-0.017 ± 0.031	8/31	0.493 ± 0.014
loops	union	↓	-0.019 ± 0.077	31/31	0.489 ± 0.030
duplicated_code_lines	union	↓	-0.021 ± 0.073	31/31	0.492 ± 0.029
avg_fan_out	total	↓	-0.025 ± 0.079	25/31	0.495 ± 0.032
max_lcom4	total	↓	-0.026 ± 0.026	4/31	0.489 ± 0.011
max_dit	union	↓	-0.027 ± 0.024	8/31	0.488 ± 0.010
distance_main_sequence	focus	↓	-0.031 ± 0.055	17/31	0.486 ± 0.023
avg_function_length	focus	↓	-0.033 ± 0.079	31/31	0.496 ± 0.030
loops	total	↓	-0.034 ± 0.075	31/31	0.484 ± 0.029
duplicated_code_lines	total	↓	-0.036 ± 0.070	31/31	0.485 ± 0.028
non_cohesive_classes	focus	↓	-0.037 ± 0.046	14/31	0.483 ± 0.019
avg_fan_out	focus	↓	-0.039 ± 0.070	22/31	0.485 ± 0.029
max_lcom4	union	↓	-0.041 ± 0.045	12/31	0.484 ± 0.019
avg_dit	focus	↓	-0.046 ± 0.051	19/31	0.484 ± 0.020
max_fan_out	total	↓	-0.052 ± 0.035	5/31	0.476 ± 0.015
type_annotation_coverage	total	↑	-0.082 ± 0.078	28/31	0.474 ± 0.031
avg_rfc	total	↓	-0.084 ± 0.087	31/31	0.463 ± 0.032
halstead_difficulty	total	↓	-0.107 ± 0.079	31/31	0.457 ± 0.031
single_char_identifier_ratio	total	↓	-0.127 ± 0.082	31/31	0.443 ± 0.032
type_annotation_coverage	union	↑	-0.133 ± 0.080	26/31	0.455 ± 0.031
non_cohesive_classes	total	↓	$-0.142 \pm 0.047^{*}$	23/31	0.443 ± 0.019
non_cohesive_classes	union	↓	$-0.145 \pm 0.048^{*}$	22/31	0.442 ± 0.019
avg_maintainability_index	union	↑	-0.148 ± 0.083	31/31	0.448 ± 0.032
avg_line_length	union	↓	-0.151 ± 0.087	31/31	0.446 ± 0.034
avg_line_length	focus	↓	-0.163 ± 0.083	31/31	0.438 ± 0.031
halstead_effort	total	↓	$-0.179 \pm 0.072^{*}$	31/31	0.435 ± 0.030
halstead_effort	union	↓	$-0.182 \pm 0.071^{*}$	31/31	0.435 ± 0.028
avg_line_length	total	↓	$-0.204 \pm 0.083^{*}$	31/31	0.426 ± 0.032
median_function_length	focus	↓	$-0.207 \pm 0.062^{*}$	28/31	0.417 ± 0.025
single_char_identifier_ratio	union	↓	$-0.209 \pm 0.075^{*}$	31/31	0.418 ± 0.028
avg_maintainability_index	total	↑	$-0.262 \pm 0.062^{*}$	28/31	0.395 ± 0.025
branches	union	↓	$-0.266 \pm 0.083^{*}$	31/31	0.397 ± 0.034
branches	total	↓	$-0.280 \pm 0.078^{*}$	31/31	0.389 ± 0.032
halstead_bugs	union	↓	$-0.358 \pm 0.065^{*}$	31/31	0.365 ± 0.027
halstead_bugs	total	↓	$-0.384 \pm 0.061^{*}$	31/31	0.350 ± 0.025

I ADDITIONAL FIGURES

These figures mirror Figures 1 and 2: leaderboards (Figures 3 and 4), passing versus failing completions (Figures 5 and 6). Each view is shown for the SWE-Quality Aligned Index (Extra Trees) and then for *Overall*, beside one panel per family. Rows keep the index's order in every chart. Correctness is the pass rate on DeepSWE's 33 Python tasks under its own scoring, and the index is scored only on the 3,145 completions the judge never saw.

I.1 LEADERBOARDS

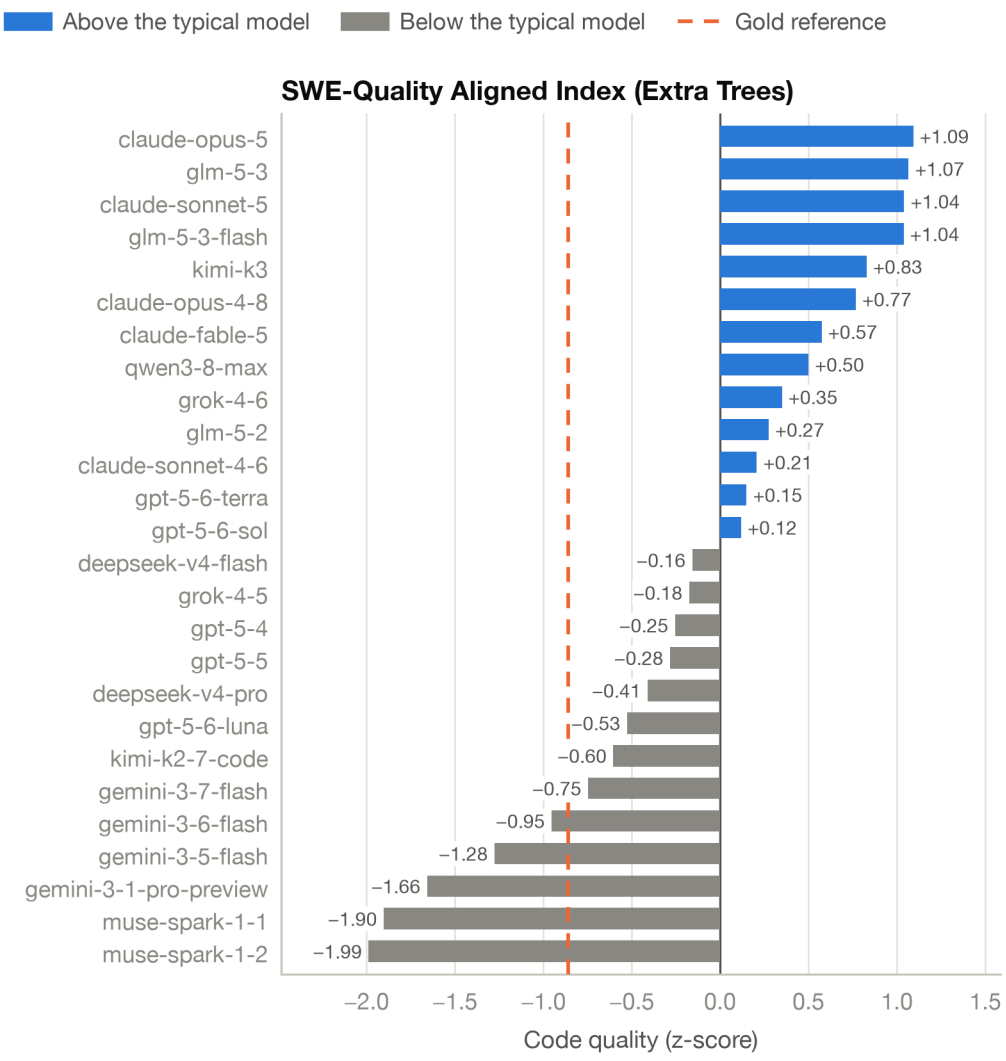


Figure 3: **The SWE-Quality Aligned Index (Extra Trees) by model**, scored on the unjudged completions. Values are z-scores. The dashed line marks the gold reference patches.

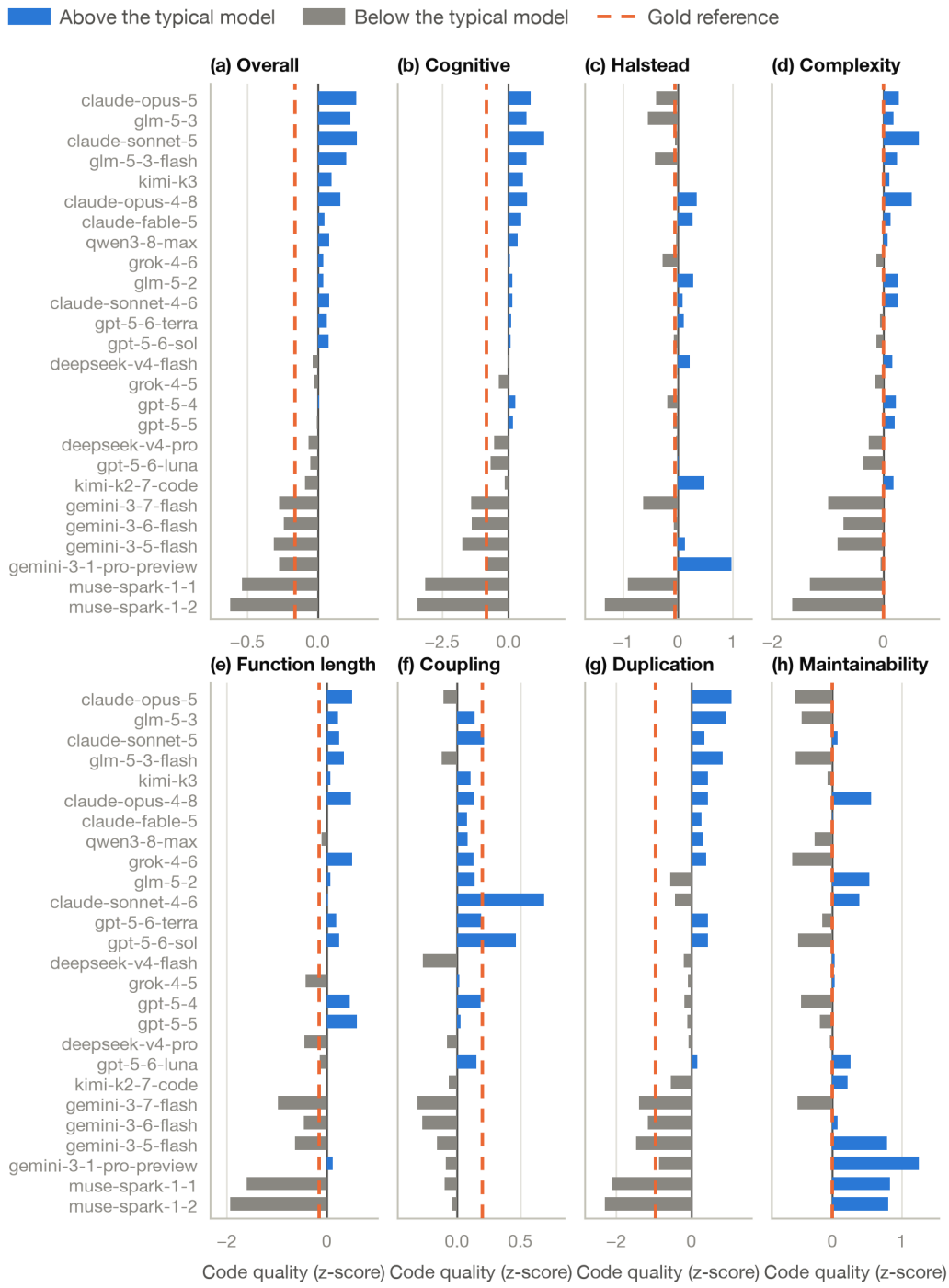


Figure 4: **Code quality by model, overall and by metric family**, as in Figure 2. **(a)** *Overall*. **(b)–(h)** The most established directional metric of seven families, averaged over the three lenses. The gold reference sits below the typical model overall and on cognitive complexity, Halstead difficulty, function length and duplication, at it on complexity and the maintainability index, and above it on coupling.

I.2 PASSING VERSUS FAILING COMPLETIONS

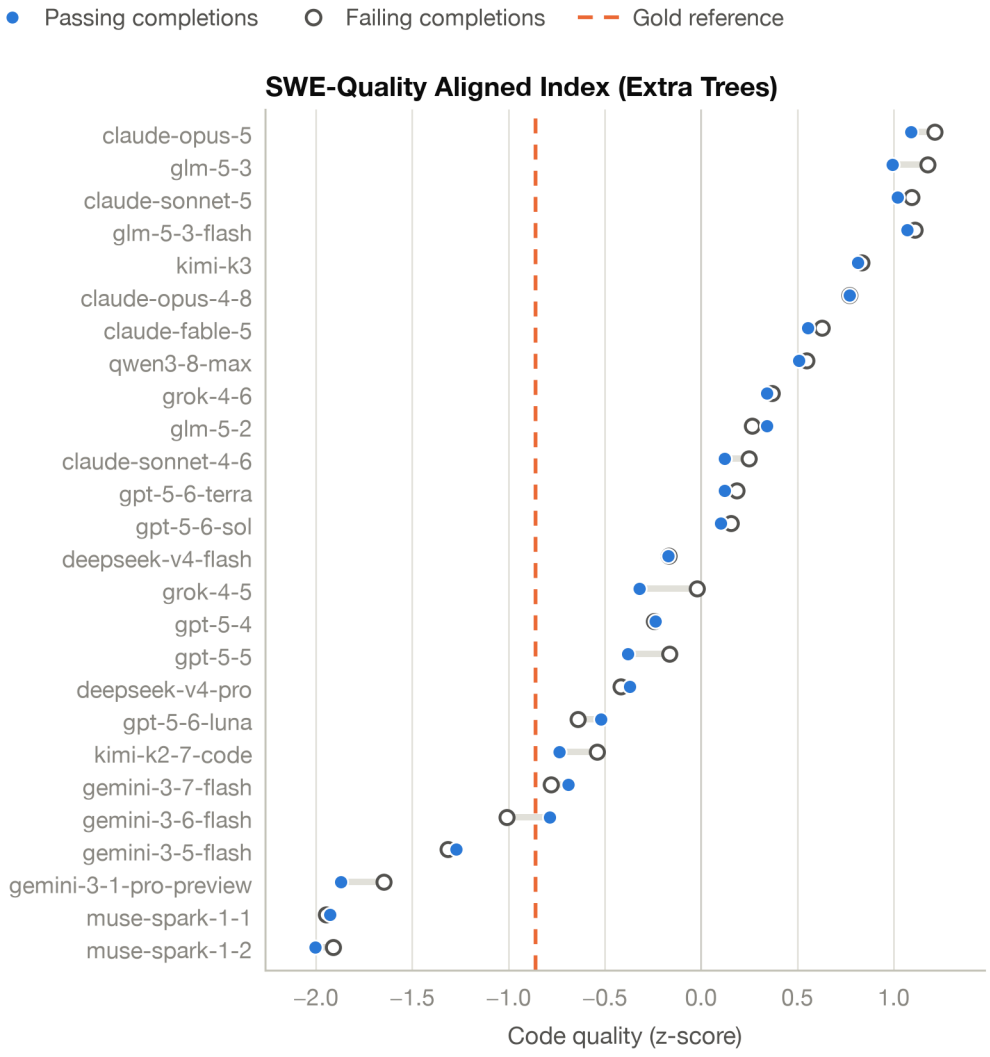


Figure 5: **Passing versus failing completions under the SWE-Quality Aligned Index (Extra Trees)**, scored only on the unjudged completions. Each model's score is computed separately over its passing (filled) and its failing (open) completions, both standardised against the same per-task reference as the all-completion score. Trials DeepSWE does not score count as neither. The two agree across models ($\rho = +0.99$), and passing completions score lower by 0.05 standard deviations (higher for 8 of 26 models; Wilcoxon $p = 0.06$, or $p = 0.47$ on tasks with both outcomes).



Figure 6: **Passing versus failing completions, overall and by metric family**, for the panels of Figure 2, computed as in Figure 5. **(a)** *Overall*. The two agree across models (Spearman $\rho = +0.97$; $+0.98$ using only tasks on which the model both passes and fails), and passing completions are slightly less clean, by 0.02 standard deviations on average (cleaner for 10 of 26 models; Wilcoxon $p = 0.10$). **(b)–(h)** The two agree across models for every family, at $\rho = +0.73$ to $+0.97$ for six families and least for coupling ($\rho = +0.55$). On every family passing completions are cleaner for between 10 and 13 of the 26 models, and the average difference is at most 0.06 standard deviations.