RuleLedger v3: Measuring Reasoning Effort in LLM Software Engineering

A controlled benchmark for GPT-5.5 reasoning-effort studies

By Adam Owada, with Codex

June 17, 2026

Abstract

RuleLedger v3 is a controlled software-engineering benchmark for measuring whether larger reasoning budgets produce better code on a realistic repair-and-extension task. The benchmark asks a model to implement one coherent subscription-ledger engine across TypeScript and Python, bitemporal replay, account lineage, corrections and voids, exact billing, deterministic reports, performance constraints, and maintainable architecture.

This paper reports a completed 200-run GPT-5.5 study: 50 implementation runs each at `low`, `medium`, `high`, and `xhigh` reasoning. The primary endpoint is aggregate quality under the `reasoning_ladder_v3` scoring profile, a 0 to 1 composite weighted toward hidden correctness and cross-language parity. Mean quality was `0.434` for low, `0.462` for medium, `0.696` for high, and `0.755` for xhigh.

The central result is the jump from medium to high reasoning. High exceeded medium by `+0.234` quality points with an approximate 95% interval of `[+0.161, +0.308]`. High and xhigh also strongly outperformed low and medium as groups. Xhigh had the best observed mean quality, hidden correctness, judge score, parity, performance, and tail behavior; no xhigh run fell below `0.4` quality. The adjacent xhigh-over-high mean gap was smaller: `+0.059` with an approximate 95% interval of `[-0.014, +0.132]`.

For RuleLedger v3-like coding tasks, high reasoning marks the quality/cost knee: it captures the main quality transition at materially lower token and wall-clock cost than xhigh. Xhigh has the best observed mean quality and weak-run reduction. The supported benchmark claim is precise: RuleLedger v3 strongly differentiates high/xhigh reasoning from low/medium reasoning, and it strongly differentiates high from medium. Medium-over-low and xhigh-over-high are directional adjacent results rather than settled separations.

Terminology

Term	Meaning
RuleLedger v3	A synthetic but realistic subscription-ledger repair benchmark with TypeScript and Python implementation surfaces.
GPT-5.5	The implementation model used for all measured runs and the judge model used for scoring.
Reasoning level	A runtime reasoning-effort setting in the benchmark harness: `low`, `medium`, `high`, or `xhigh`.
Hidden correctness	Private RuleLedger v3 behavior checks excluding parity and performance categories.
Hidden parity	Private checks that TypeScript and Python outputs agree on equivalent cases.
Judge score	A GPT-5.5 xhigh review signal over source, diffs, logs, public checks, and hidden-result summaries.
Aggregate quality	The weighted 0 to 1 composite score used as the primary endpoint.
Implementation tokens	GPT-5.5 implementation tokens parsed from `codex exec --json` event streams. Judge tokens are excluded from cost tables.
Tail behavior	The distribution of weak and strong runs, especially threshold counts such as runs below `0.4` or above `0.8` quality.

Benchmark Task

RuleLedger v3 is a subscription-ledger implementation benchmark. The visible prompt is an issue brief rather than a complete truth table. Public tests are readable and intentionally incomplete. Hidden tests evaluate whether the model generalizes across behavior families while preserving earlier compatibility behavior.

The measured agent starts from a frozen starter project with both TypeScript and Python implementation surfaces. A strong solution must normalize event streams, replay account state, compute billing and entitlements, export deterministic reports, and preserve compatibility with earlier public APIs. The two language implementations must agree.

The hard part is coherence. A run can make a summary pass while CSV output remains wrong, fix TypeScript while Python drifts, or handle corrections before account merges but fail after lineage changes. RuleLedger v3 rewards implementations that build one canonical replay model and apply it across summaries, reports, parity outputs, replay digests, and performance-sensitive paths.

Benchmark Design

Axis	What it tests	Why it challenges reasoning
Bitemporal replay	Audit-visible and business-effective cutoffs, corrections, voids, and deterministic ordering	Requires temporal reasoning before state application
Account lineage	Merges, old identifiers, target resolution, and post-merge actions	Requires durable alias modeling across event history
TypeScript/Python parity	Comparable public outputs, CSV serialization, replay digests, and edge-case normalization	Penalizes language-specific shortcuts
Exact billing	Proration, money parsing, rounding, invoice dates, and large seat counts	Exposes imprecise numeric handling
Regression pressure	v1/v2 public APIs and pass-to-pass behavior remain in scope	Prevents narrow fixes that break earlier contracts
Performance	Large generated ledgers and replay/report workloads	Catches nested scans and repeated full-history recomputation
Maintainability	Multi-module architecture with thin compatibility boundaries	Rewards code that reviewers can reason about

Benchmark Development

RuleLedger v3 came out of a longer calibration process. The early RuleLedger benchmark established the harness and the subscription-ledger domain, but public tests, typecheck, parity, reporting, normalization, and minimality often saturated. Most of the useful signal came from a small number of hidden parse-validation and state-reduction cases.

RuleLedger v2 expanded the semantic surface. It added bitemporal business/audit views, lifecycle precedence, account merges, proration, performance cases, category-level reporting, and v2-specific public hooks. That made the task more realistic, but capable models could still localize the visible rule list, patch the obvious TypeScript and Python files, and score well without much repo-scale synthesis.

RuleLedger v3 changed the target from rule following to system integration. The starter became a multi-module TypeScript/Python project with public issue-style requirements, hidden fail-to-pass and pass-to-pass checks, metamorphic cases, cross-language parity, performance workloads, and a judge signal. The goal was to measure whether a model could infer one ledger semantics model and carry it through parsing, replay, billing, reports, compatibility APIs, and both language implementations.

The v3 development process required several calibration passes before the benchmark produced reasoning-level separation:

Iteration	Calibration lesson	Resulting change
v1	Too many components saturated; low and medium runs could often patch locally	Keep the domain, but increase hidden semantic depth and version the benchmark assets
v2	Harder rules helped, but the visible work remained easy to localize	Move from explicit rule lists toward an issue-resolution task with cross-module pressure
Early v3	Larger hidden fixtures made the task harder without reliably separating reasoning levels	Split integrated incidents into staged diagnostic checks with partial-credit surfaces
Mid v3	Strong runs sometimes refactored in the right direction but were penalized as too large or missed one opaque fixture	Reweight localization/module-ownership signals, lower minimality pressure, and enforce structured judge output
Final v3 calibration	Separation appeared only after the task made cross-surface synthesis explicit	Add a Resolution Standard: credible fixes should use one canonical replay model across summaries, CSV reports, TypeScript/Python parity, replay digests, billing, and module ownership

The calibration pass that finally separated reasoning levels used same-ledger, multi-view pressure. A single incident was checked through point-in-time summaries, CSV reports, parity outputs, replay digests, performance behavior, and compatibility tails. That exposed the difference between local patching and durable implementation design. In the finished benchmark, medium often makes local progress, while high and xhigh more reliably build and preserve a coherent model across the whole task.

Experiment Design

The official study contains `200` scored implementation runs:

Cell	Reasoning	Model	Runs
V3P0	low	GPT-5.5	50
V3P1	medium	GPT-5.5	50
V3P2	high	GPT-5.5	50
V3P3	xhigh	GPT-5.5	50

Measured implementation runs used `codex exec --json`. Raw JSONL events, prompts, rendered configs, stderr logs, diffs, test logs, judge output, metadata, scores, HTML reports, and PDF reports were preserved under run-specific directories. The implementation model was GPT-5.5 in all cells. The judge was GPT-5.5 xhigh in all cells.

The study accumulated repeats in six measured batches. Pooling required the full expanded matrix hash, selected repeat ranges, benchmark template, hidden cases, prompts, scoring config, judge schema, model identifiers, timeout policy, and failure policy to remain fixed. All six measured batches passed final validation. There were zero implementation failures and zero judge failures.

Measurement

Reported values use fixed rounding: quality-like scores, deltas, intervals, and component means are rounded to three decimals; token counts are shown as whole-token means or `M` shorthand in prose; efficiency ratios use three significant figures in scientific notation.

Quality

Quality is a 0 to 1 composite score from `configs/scoring_v3.yaml`:

Component	Weight	Meaning
Hidden correctness	0.50	Correctness on frozen private RuleLedger v3 cases, excluding parity and performance categories
Hidden parity	0.15	TypeScript/Python agreement on hidden parity checks
Performance	0.10	Performance behavior on hidden workloads
Judge	0.22	GPT-5.5 xhigh assessment of source, diffs, logs, public checks, and hidden-result summaries
Minimality	0.03	Production LOC minimality signal relative to the configured target

The weighting intentionally places most emphasis on hidden behavioral correctness while preserving signals from parity, performance, expert-style judge review, and implementation size. Minimality is deliberately small so a compact but incomplete patch cannot dominate a correct and maintainable implementation.

The judge was fixed across conditions and contributes to a broader scoring profile anchored mostly in hidden tests. Its output is a consistent expert-style review signal; independent human adjudication remains outside this study.

Tokens and Time

The harness parsed implementation usage from `codex exec --json` event streams. Because every measured implementation run used GPT-5.5, implementation tokens are GPT-5.5 implementation tokens. Judge tokens are excluded from the cost table because the measured developer decision is which implementation reasoning level to run.

Elapsed seconds are per-run harness measurements. They compare conditions within this study, with local scheduling, batch parallelism, and execution environment as confounders.

Uncertainty

The paper reports observed sample means, standard deviations, approximate confidence intervals, pairwise mean-difference intervals, and descriptive effect sizes. These are not preregistered inferential tests. Small adjacent gaps should be read cautiously, especially when intervals cross zero.

Results at a Glance

Reasoning	Runs	Quality mean	Quality sd	Hidden correctness	Judge	Impl tokens	Impl seconds	Primary read
low	50	0.434	0.181	0.364	0.394	815,190	199	Useful local progress, weak global consistency
medium	50	0.462	0.175	0.409	0.459	1,389,968	318	Modest gain over low
high	50	0.696	0.195	0.674	0.683	2,612,154	686	Main quality jump
xhigh	50	0.755	0.171	0.739	0.752	3,333,886	1,019	Best mean and tail behavior

Detailed Findings

Low Reasoning

Low reasoning scored `0.434` mean quality. It produced some successful runs (`5/50` at or above `0.7` quality), but `20/50` runs fell below `0.4`. The pattern is consistent with local repair ability without reliable global integration.

Low serves as a lower-bound condition for benchmark calibration. For RuleLedger v3-style implementation work where quality matters, high and xhigh are the relevant settings.

Medium Reasoning

Medium reasoning scored `0.462`, only `+0.028` above low. The approximate 95% interval for medium minus low was `[-0.043, +0.099]`, so the study supports only a weak adjacent-separation claim between those two levels.

Medium used substantially more tokens than low (`1.390M` vs `0.815M`) while producing only a modest mean quality gain. In this benchmark, medium did not represent the quality/cost knee.

High Reasoning

High reasoning scored `0.696`, a `+0.234` gain over medium. This is the dominant empirical result. The approximate 95% interval for high minus medium was `[+0.161, +0.308]`, and the descriptive effect size was large (`d = 1.263`).

High is where the measured quality curve changes. It costs materially more than medium, but it is the first setting with consistently higher hidden correctness and fewer weak runs.

Xhigh Reasoning

Xhigh reasoning scored `0.755`, the best observed mean. It also had the best hidden correctness, judge score, parity, performance, and tail behavior. No xhigh run fell below `0.4` quality, compared with `3/50` for high, `17/50` for medium, and `20/50` for low.

The adjacent xhigh-over-high gap was `+0.059`, with an approximate 95% interval of `[-0.014, +0.132]`. The evidence is directional; the high-over-medium result is the stronger adjacent-separation finding.

Pairwise Quality Differences

Comparison	Mean difference	Approx. 95% interval	Cohen's d	Interpretation
medium - low	+0.028	[-0.043, +0.099]	0.158	Small, not robustly separated
high - medium	+0.234	[+0.161, +0.308]	1.263	Large, robust adjacent separation
xhigh - high	+0.059	[-0.014, +0.132]	0.322	Positive mean, interval crosses zero
high - low	+0.262	[+0.188, +0.337]	1.395	Large separation
xhigh - low	+0.321	[+0.251, +0.391]	1.824	Very large separation
xhigh - medium	+0.293	[+0.224, +0.362]	1.691	Very large separation

This is the paper's main claim boundary. RuleLedger v3 strongly separates high/xhigh from low/medium. It strongly separates high from medium. The medium-over-low and xhigh-over-high observed means point in the expected direction, but they do not carry the same evidentiary weight.

Secondary Metrics

Reasoning	Hidden tests	Hidden correctness	Hidden parity	Performance	Judge
low	0.379	0.364	0.531	0.556	0.394
medium	0.420	0.409	0.506	0.515	0.459
high	0.687	0.674	0.734	0.738	0.683
xhigh	0.754	0.739	0.791	0.800	0.752

Hidden-test and judge signals show the same broad ladder: low and medium are clustered together, high improves sharply, and xhigh leads high on mean.

Behavior-Family Results

Hidden categories should be interpreted as behavior-family aggregates rather than public fixture descriptions:

Category	low	medium	high	xhigh
evolution	0.314	0.295	0.562	0.605
fail_to_pass	0.420	0.431	0.645	0.706
localization	0.134	0.324	0.773	0.875
metamorphic	0.385	0.369	0.552	0.600
parity	0.531	0.506	0.734	0.791
pass_to_pass	0.780	0.800	0.865	0.933
performance	0.556	0.515	0.738	0.800

The largest visible differences are in localization, staged evolution, parity, and performance. That pattern matches the benchmark design: lower-reasoning runs can often make local fixes, but higher-reasoning runs are better at maintaining one implementation model across interacting requirements and language surfaces.

Tail Behavior

Reasoning	>=0.4 quality	>=0.5 quality	>=0.7 quality	>=0.8 quality	>=0.9 quality	<0.4 quality
low	30/50	15/50	5/50	4/50	0/50	20/50
medium	33/50	20/50	2/50	1/50	1/50	17/50
high	47/50	42/50	24/50	18/50	9/50	3/50
xhigh	50/50	48/50	26/50	22/50	14/50	0/50

Xhigh's advantage extended beyond mean quality. It also removed the low-quality tail seen in the other settings. For engineering use, avoiding bad outputs can matter as much as raising the average score.

Cost and Latency

Quality per GPT token in this table uses ratio-of-means: mean quality divided by mean implementation tokens.

Reasoning	Mean implementation tokens	Mean implementation seconds	Quality mean	Quality per GPT token
low	815,190	199	0.434	5.32e-7
medium	1,389,968	318	0.462	3.32e-7
high	2,612,154	686	0.696	2.66e-7
xhigh	3,333,886	1,019	0.755	2.26e-7

Raw quality per token falls as reasoning increases because low-reasoning runs are cheaper. That metric over-rewards cheap low-quality runs when the implementation has to clear a quality threshold. The useful comparison is quality achieved, weak-run risk, and cost. High delivers the main quality jump. Xhigh costs more and has lower raw quality per token, while producing the best observed mean and strongest tail behavior.

Why High Was the Main Jump

RuleLedger v3 combines local coding work with global consistency obligations. Low and medium runs often pass public tests and produce complete artifacts, but they more frequently leave independent replay paths, parity drift, incomplete merge-lineage handling, localization failures, or performance weaknesses.

High reasoning appears to cross the threshold where the model more often infers the required abstraction: one shared replay model that drives summaries, reports, parity outputs, and digests. Once that abstraction is in place, many hidden behavior families improve together.

Why Xhigh Helped Less Than High

Xhigh produced the best mean and tail behavior, but the marginal gain over high was smaller than the high-over-medium jump. High reasoning already solves much of the abstraction problem in this benchmark.

The xhigh result remains important. It reduced weak-run risk and led every secondary mean. The data support xhigh as the best observed mean and tail-risk setting, while leaving robust xhigh-over-high separation unresolved.

Interpretation

RuleLedger v3 evaluates whether a coding model can maintain a coherent implementation across interacting software constraints. The task punishes narrow local fixes and rewards shared semantics across language surfaces, reports, billing, parity, and performance behavior.

The 200-run result identifies high reasoning as the main transition point. High captures most of the quality jump while costing materially less than xhigh.

Xhigh had the best observed mean and no runs below `0.4` quality. Its advantage over high appears in tail behavior and secondary means, not in a settled adjacent-separation interval.

Frame adjacent reasoning claims carefully. The study strongly supports high over medium. It supports xhigh as the best observed mean and best tail-risk setting. Medium-over-low and xhigh-over-high carry less evidentiary strength.

For future benchmark work, add preregistered statistical tests, distribution plots, batch-stratified plots, a failure-mode taxonomy from sampled diffs, and an independent human or non-LLM review layer for maintainability-sensitive claims.

Limitations

- Scope is one main implementation model family and four reasoning settings.
- The judge is another LLM held fixed across conditions; independent human review remains future work.
- The benchmark is synthetic, even though it is designed to mimic realistic ledger engineering work.
- Hidden tests are finite and may not capture all semantically valid implementation strategies.
- Batch-level variance is visible. Small batches can invert adjacent reasoning levels.
- The execution environment used high local parallelism and `danger-full-access` sandboxing, which should be documented when comparing against other environments.
- Token and elapsed-time metrics are implementation-cost signals. They are not price or universal latency estimates.
- The statistical analysis is descriptive: approximate confidence intervals and effect sizes, without a preregistered inferential analysis plan.

Within those limits, the main result remains: RuleLedger v3 exposes a large quality transition between medium and high reasoning in GPT-5.5 software-engineering work.

Appendix A: Batched Execution

Batch	Repeat range	Runs	Started UTC	Finished UTC	Validation	Implementation failures	Judge failures
v3-paper-batch-001	1-5	20	2026-06-05T19:05:14	2026-06-05T20:00:48	passed	0	0
v3-paper-batch-002	6-10	20	2026-06-05T22:20:18	2026-06-05T23:17:05	passed	0	0
v3-paper-batch-003	11-20	40	2026-06-10T08:11:40	2026-06-10T09:58:36	passed	0	0
v3-paper-batch-004	21-30	40	2026-06-10T17:10:24	2026-06-10T18:47:24	passed	0	0
v3-paper-batch-005	31-40	40	2026-06-11T05:37:56	2026-06-11T07:25:00	passed	0	0
v3-paper-batch-006	41-50	40	2026-06-17T21:11:06	2026-06-17T22:49:53	passed	0	0

The first three batches were launched from a clean repository. Batches 4-6 recorded `repo.dirty: true` because the untracked `papers/` artifact already existed. The measured repository head stayed fixed across all six batches:

`2cd7314b10cacaf8b1b3089aa9febdd081b328bf`

Appendix B: Per-Batch Means

Batch	Reasoning	Runs	Quality	Hidden correctness	Judge
batch 001	low	5	0.475	0.401	0.451
batch 001	medium	5	0.590	0.547	0.583
batch 001	high	5	0.527	0.490	0.429
batch 001	xhigh	5	0.674	0.630	0.703
batch 002	low	5	0.378	0.294	0.348
batch 002	medium	5	0.496	0.498	0.535
batch 002	high	5	0.865	0.863	0.832
batch 002	xhigh	5	0.798	0.794	0.780
batch 003	low	10	0.463	0.409	0.432
batch 003	medium	10	0.484	0.428	0.424
batch 003	high	10	0.802	0.787	0.779
batch 003	xhigh	10	0.852	0.850	0.838
batch 004	low	10	0.360	0.273	0.366
batch 004	medium	10	0.364	0.298	0.377
batch 004	high	10	0.636	0.622	0.655
batch 004	xhigh	10	0.782	0.776	0.778
batch 005	low	10	0.509	0.436	0.449
batch 005	medium	10	0.351	0.274	0.369
batch 005	high	10	0.677	0.650	0.668
batch 005	xhigh	10	0.655	0.609	0.702
batch 006	low	10	0.410	0.353	0.323
batch 006	medium	10	0.568	0.524	0.566
batch 006	high	10	0.670	0.636	0.684
batch 006	xhigh	10	0.750	0.749	0.701

The per-batch table explains why small pilots were informative but insufficient. Batch 001 made high look weaker than medium. Batch 002 made high look stronger than xhigh. Batch 005 again had xhigh below high on

aggregate quality. The completed pooled sample is what stabilizes the broader result.

Appendix C: Reproducibility

Key source files:

- Experiment config: `configs/ruleledger_v3_paper_50.yaml`
- Scoring profile: `configs/scoring_v3.yaml`
- Issue brief: `benchmark_template_v3/docs/ruleled ger_v3_issue_brief.md`
- Architecture notes: `benchmark_template_v3/docs/ruleled ger_v3_architecture.md`
- Development plan: `plans/stage-22-ruleledger-v3.md`
- Batched repeat requirements: `plans/batched-repeat-study-requirements.md`

Completed measured batches:

- `runs/20260605T190514-ruleledger_v3 _paper_50-v3_paper_batch_01_r01_r0 5_measured`
- `runs/20260605T222018-ruleledger_v3 _paper_50-v3_paper_batch_02_r06_r1 0_measured`
- `runs/20260610T081140-ruleledger_v3 _paper_50-v3_paper_batch_03_r11_r2 0_measured`
- `runs/20260610T171024-ruleledger_v3 _paper_50-v3_paper_batch_04_r21_r3 0_measured`
- `runs/20260611T053756-ruleledger_v3 _paper_50-v3_paper_batch_05_r31_r4 0_measured`
- `runs/20260617T211106-ruleledger_v3 _paper_50-v3_paper_batch_06_r41_r5 0_measured`

Shared frozen hashes:

- Full matrix SHA-256: `2ec216ba7cf6d9e01528a355ac590aec7a edf8b026c9aa8d49f49fdd623a27e1`
- Benchmark template SHA-256: `edfde6362042bfdf100bedfe63c273c817 072004cc42ff3952ad3afb9c319699`
- Hidden cases tree SHA-256: `bf742505d0503883a3e8dc001d41a16254 b67d1aa3aea1fd337acb5e620a8d6b`
- Hidden manifest SHA-256: `63937005528b0ff1842bcd10a57617da49 acbbbc0e4b67dfa8b5f65547ecb362`
- Scoring config SHA-256: `dc50227fd818efc567b756bc2be2bf6bf2 a22850072bd7ab14ca6673c68cb76c`
- Experiment config SHA-256: `2f47b30ad9763cebb332fcbcef60488b0d aab479c2aec2765f447d0af7d38aa3`
- Repository head for measured source: `2cd7314b10cacaf8b1b3089aa9febdd081b328bf`

Run directories preserve raw JSONL events, stderr logs, prompts, rendered configs, diffs, test logs, judge output, metadata, scores, and generated HTML and PDF reports.

Appendix D: References

- RuleLedger v3 issue brief: `benchmark_template_v3/docs/ruleled ger_v3_issue_brief.md`
- RuleLedger v3 architecture notes: `benchmark_template_v3/docs/ruleled ger_v3_architecture.md`
- RuleLedger v3 development plan and calibration log: `plans/stage-22-ruleledger-v3.md`
- Batched repeat study requirements: `plans/batched-repeat-study-requirements.md`
- SWE-bench repository: https://github.com/swe-bench/SWE-bench
- SWE-bench Verified announcement: https://openai.com/index/introduci ng-swe-bench-verified/

- SWE-Bench Pro: https://openreview.net/forum?id=6RYawev6L9
- SWE-Lancer: https://openai.com/index/swe-lancer/