js262-faceoff: Frontier Coding Models Writing JavaScript Engines in C from Scratch, Scored by test262

Seconds_0
with Claude*

June 2026

Abstract

We present *js262-faceoff*, a single-session agentic coding evaluation in which frontier language models, each driven headlessly by its own CLI agent on an isolated cloud machine, received an identical prompt: write a JavaScript (ECMAScript) engine in C from scratch—C11/C17, libc and libm only, no third-party code, no porting of existing engines—within eight hours plus a one-hour wrap-up. The final committed artifact is scored by the number of official tc39/test262 conformance cases passed (91,280 cases at a pinned revision) under a hardened reference runner with anti-gaming staging, secret computed-output probes, sandboxed execution, and an AddressSanitizer-clean primary metric.

Three contestants ran under an equal default harness: Claude Opus 4.8, Claude Opus 4.7, and GPT-5.5. Claude Opus 4.8 won this canonical comparison at 88.9%, followed by Opus 4.7 at 65.1% and GPT-5.5 at 45.7%—the last of which also silently mis-executed all six workloads of an observational benchmark that it compiled cleanly. A maximum-effort parallel-workflow harness variant raised Opus 4.8 to 92.8%—above the mature QuickJS-NG engine on the same denominator—while spending a quarter of the output tokens of its own default run. After the v1 draft was complete, Claude Fable 5 (`claude-fable-5`)—a newer model generation than the round-1 contestants—was evaluated in a two-run addendum under the identical protocol, hardware, and pins: its default-harness run reached 97.7%—the strongest greenfield build result in the study, 11.5 points above QuickJS-NG on this denominator—while emitting the fewest novel tokens of any greenfield Claude run, and its workflow variant reached 96.8%. Across the two models run under both harnesses, the workflow variant helped exactly one of the two, indicating that harness gains are model-dependent rather than a free improvement. Further add-on runs (the Fable 5 workflow variant and an incomplete Gemini 3.5 Flash run) are reported separately.

A third round restarted a field of returning contestants from the same champion engine and raced them to fix its remaining 1,865 reachable failures under a no-regression gate: five of the eight runs were disqualified by the gate, every one after relying on sampled gate checks, while the three survivors were all Claude-family runs: the winner was the only contestant to complete the full-suite gate (launched detached to survive the harness turn budget), and the other two survived by keeping tested and shipped code identical; the parallel-workflow harness, which had not helped Fable 5 in the greenfield round, converted its speed advantage into a gate-clean win only for that model—both Opus workflow variants regressed—indicating that orchestration amplifies whatever discipline the model already has. A fourth round then raced

*The judge harness, run orchestration, scoring infrastructure, and manuscript preparation were operated with Claude (Claude Code) [1]. The contestant runs themselves were fully autonomous; no human or operator assistance was given to any contestant after its initial prompt.

the same field to make the champion engine as fast as possible, scored by geometric-mean speedup on a hidden regenerated workload suite under the same zero-regression gate: six of eight runs shipped gate-clean, the workflow harness took its second straight win (Opus 4.8 at 4.79x), and the only zeroed runs were the two Claude Opus 4.7 variants—one of them the fastest raw engine in the field at 5.11x—leaving that family 0-for-4 on gates across the two gated rounds. A fifth round re-ran the speed race as a blind replication—identical baseline, brief, budgets, and field, on boxes carrying no trace of the first running—with one intended change, a revised system prompt (an alternate variant of the harness's production system prompt) on the two Fable 5 runs: the variant's workflow run won at a series-record gated 5.19x behind a two-full-gate endgame, the round-4 champion configuration repeated its sampled-evidence-only strategy verbatim and was zeroed by a two-case regression, the Opus 4.7 family fell to 0-for-6 on gates, and the six unchanged controls swung on identical workloads by as much as—and in one case roughly twice—the treatment effect, bounding what any single-run comparison in the series can claim. We document the full methodology, every operational incident and fairness ruling, and per-run effort, token, and parallelism profiles.

1 Introduction

Most public coding evaluations measure one of two regimes: short, well-specified problems such as function synthesis or unit-test repair [3, 8], or long-horizon work mediated by a human who reviews, redirects, and rescues the model. This work evaluated a third regime: a single autonomous session, on a task far larger than any human could complete in the same wall-clock time, with an objective external score that the agent could not control or game by construction. Each contestant model was given the same prompt, an identical isolated machine, and an equal nominal budget of 8 hours of work plus a 1-hour wrap-up, and was asked to build a JavaScript engine in C from scratch. The final committed binary was then scored against the official `tc39/test262` conformance suite [7] by a hardened reference runner that the contestant never saw.

This task has several properties that make it unusually well suited to measuring sustained autonomous engineering.

- **Objective and external.** The score is the number of test262 cases passed by the final committed binary under the judge's runner. There is no rubric, no judge model, and no partial credit for prose.
- **A large, hard-to-game denominator.** The scored manifest contains 91,280 cases at a pinned corpus revision (SHA `4249661388e5d3f92a85186213da140a6481490f`), expanded per the suite's interpretation rules [6], and is fixed for every contestant. An engine that does nothing scores zero: before a completion-sentinel was added to the staging pipeline, a do-nothing exit-0 stub passed roughly 79% of the corpus (essentially every positive non-async case); after it, the same stub scores zero (Section 4). An engine that fakes results is caught by secret computed-output probes regenerated fresh at each scoring run. Marginal score requires marginal real semantics.
- **Far beyond a few-hour human task.** A working ECMAScript engine [5] spans lexing, parsing, an object model, an evaluator, a standard library, regular expressions, Unicode, async execution and job queues, and modules. Production engines represent years to decades of engineering [2]. Even a partial pass at the corpus stresses sustained planning, prioritization, self-measurement, and regression control over a full working day of unattended operation—capabilities that short benchmarks cannot observe.

- **Naturally graded difficulty.** The corpus rewards breadth (everyday builtins), depth (parser early errors, property descriptors, completion semantics), and strategic triage. Different models can reach similar scores by different routes, and the route is itself informative.
- **Resistant to contamination.** test262 is public, but the contest score is produced by a runner with stripped test metadata, randomized filenames, uniform camouflage trailers on every non-raw staged test, and secret probes generated at scoring time, so memorized test expectations confer no advantage over implementing the language.

The headline comparison is the canonical three: Claude Opus 4.8, Claude Opus 4.7, and GPT-5.5, each run once under the default harness on identical Hetzner cpx31 boxes. Under the primary metric (sanitizer-clean passes), the canonical three spanned 88.91% (Opus 4.8) down to 45.69% (GPT-5.5) of the 91,280-case denominator. Two round-1 add-on runs are reported separately and are not rank-comparable: Opus 4.8-ultra re-ran Opus 4.8 under a maximum-effort harness variant with an explicit parallel-workflow add-on, and Gemini 3.5 Flash completed a partial run on different hardware.

After the v1 draft of this report was complete, Claude Fable 5 (`claude-fable-5`)—a newer model generation than the round-1 contestants—was evaluated in a two-run addendum under the identical protocol, hardware, and pins: its default-harness run scored 97.71%—the strongest greenfield build result in the study—and its workflow variant 96.79%, so across the two models run under both harnesses the workflow helped exactly one. As a calibration oracle, QuickJS-NG 0.15.0 [9]—a mature production engine—scored 86.23% under the same runner with zero runner discrepancies.

This report contributes:

(i) a hardened, open methodology for scoring autonomously built JavaScript engines: a reference runner with anti-gaming staging, completion sentinels, computed-output probes, sanitizer and dependency verification [10], sandboxed execution [4], and oracle validation against two production engines and the official test262 harness;

(ii) head-to-head canonical results for three frontier models at equal harness, equal hardware, and equal nominal budget;

(iii) a harness-variant comparison on two models, measuring what a maximum-effort, parallel-workflow configuration adds to (or subtracts from) the default agent loop [1];

(iv) an effort analysis across tokens, commits, and parallelism, including a novel-tokens metric (cache-creation volume) that tracks final codebase size far better than commit count;

(v) a complete incident and fairness record: every run materially degraded by operator or harness error received a documented clean rerun or resume, and the degraded artifacts were preserved rather than silently discarded.

Section 2 situates the contest among existing evaluations. Sections 3 and 4 describe the experimental setup and the scoring methodology, including its documented deviations from strict test262 semantics. Section 5 reports results; Sections 6 and 7 discuss findings and threats to validity. Section 9 reports the third round, a last-mile race in which returning contestants restarted from a shared champion engine and fixed its remaining failures under a no-regression gate. Section 10 reports the fourth round, a speed race in which the same field optimized the resulting champion engine for raw throughput under the same gate. Section 11 reports the fifth round, a blind replication of the speed race whose single intended change was a revised system prompt (an alternate

variant of the harness's production system prompt) on the two Fable 5 runs, and which doubles as the series' first measurement of round-to-round variance. Appendices reproduce the contestant prompt (Appendix A), the incident log (Appendix B), and per-run timelines (Appendix C).

2 Related Work

Coding benchmarks. Most widely used coding evaluations operate at small granularity. Static function-synthesis benchmarks such as HumanEval [3] present self-contained problems solvable in a single completion and score the result against a handful of hand-written unit tests. Agentic repair benchmarks such as SWE-bench [8] extend this to repository-scale edits, but each task is still a bounded patch against an existing human-written codebase, resolved on a timescale of minutes to hours, and the surrounding repository, build system, and test infrastructure are supplied to the model rather than produced by it. Both families measure the ability to modify or complete human scaffolding; neither measures sustained, unsupervised construction of a large artifact from nothing.

Conformance suites as oracles. test262 [7] is the official ECMAScript [5] conformance suite and serves as the de facto correctness oracle for production JavaScript engines; its structure and metadata conventions are documented in the interpreting-test262 guide [6]. Compact independent engines such as QuickJS [2] and its actively maintained fork QuickJS-NG [9] – multi-year efforts by expert engine developers – report their conformance against the same suite, which makes test262 a natural fixed yardstick: it is large (91,280 cases in the subset scored here), externally maintained, and was not written with language models in mind. We additionally used QuickJS and QuickJS-NG as oracle engines to validate the scoring runner itself (Section 4).

This work. The contest described here differs from the above along several axes. Each model worked in one autonomous session under a fixed wall-clock budget, starting from an empty repository, with no human edits to the artifact during the run. What is scored is not a transcript or a patch but a compiled C binary, evaluated after the fact by an external runner with anti-gaming hardening (frontmatter stripping, randomized filenames, completion sentinels, and secret computed-output probes; Section 4) and with sanitizer-clean execution as the primary metric. All operator-side incidents and fairness rulings are disclosed (Appendix B). To our knowledge, no prior evaluation combines this task scale (a working language implementation), this scoring substrate (a full industry conformance suite run against the produced binary), and single-session autonomy; we present it as a case study rather than a standardized benchmark, and discuss its limitations in Section 7.

3 Experimental Setup

3.1 Task and engine contract

Every contestant received the same written brief (reproduced in Appendix A): implement a JavaScript (ECMAScript) engine in C, from scratch, in a single autonomous session, with the score defined as the number of official test262 [7] conformance cases passed under the judge's reference runner. The brief imposed five hard rules. First, the implementation language was C (C11 or C17) with dependencies limited to libc and libm — no third-party libraries, code generators, garbage-collector libraries, or bignum libraries. Second, a from-scratch rule: contestants could not read, copy, port, or adapt source code from any existing JavaScript engine

or parser (QuickJS, JerryScript, duktape, mujs, V8, JavaScriptCore, SpiderMonkey, engine262, Babel, swc, etc.), and could not invoke an existing JS engine at build time or runtime; permitted references were the ECMA-262 specification [5], the test262 repository including its harness files and `INTERPRETING.md` [6], MDN, and the contest documents. Third, a no-test-gaming rule: no logic keyed to specific test262 file names, paths, or recognizable test contents; hardcoding spec constants (e.g. `Number.MAX_SAFE_INTEGER`) was permitted, hardcoding answers to tests was not. Fourth, single session with no human assistance after the prompt. Fifth, a fixed budget of 8 hours of wall-clock time, followed by a 1-hour wrap-up turn; the final committed state of `main` in the submission repository is what was scored.

The brief disclosed the enforcement mechanisms in outline (clean-build audit, linkage audit, hidden computed-output probes, transcript review) without revealing probe contents; these are described in Section 4.

The deliverable was a git repository at `~/js262-submission` containing source code and a `Makefile` with two targets: `make`, producing `./js262` (optimized, `-O2`), and `make asan`, producing `./js262-asan` (the same engine built with `-fsanitize=address,undefined -fno-sanitize-recover=all`). Both targets had to build cleanly with clang with no warnings under `-Wall -Wextra`. A short `STATUS.md` (what is implemented, what is known missing, self-measured score) was also required.

The engine contract, as stated in the brief, was:

```
./js262 [--module] <file1.js> <file2.js> ... <fileN.js>
```

All files are evaluated in order in a single shared realm. Files $1 \ldots N-1$ are always evaluated as classic global scripts — this is how the runner supplies the test262 harness (`assert.js`, `sta.js`, etc.) — so their top-level `function` and `var` declarations must become properties of the global object. The last file is the test, evaluated as a classic script normally or as an ECMAScript module when `--module` is passed (module specifiers resolve relative to the working directory, where the runner stages fixture files).

Exit code 0 means the test ran to completion with no uncaught exception; a nonzero exit means a parse error or uncaught exception, and the first non-empty line on stderr must begin with the error constructor name followed by a word boundary (e.g. `TypeError: x is not a function`). Termination by signal (segfault, abort) always counts as a failure, even for tests that expect an error. A global `print(value)` builtin is required: it writes the string-converted value plus a newline to stdout, and is needed both for async tests and for the per-case completion sentinel (Section 4). The `$262` host object is optional, but the brief noted that implementing `$262.createRealm`, `$262.evalScript`, `$262.detachArrayBuffer`, `$262.gc`, and `$262.global` unlocks roughly 600+ additional cases; `$262.agent` (multi-threaded agents) was not expected of anyone. Per-case timeouts (3 s, retried once at 10 s, tripled for the sanitizer build) were disclosed in the brief and are detailed in Section 4.

3.2 Contestants

Seven runs were performed: five round-1 runs in two tiers, plus a two-run addendum. The *canonical three* — Claude Opus 4.8, Claude Opus 4.7, and GPT-5.5 — ran under the default harness on identical hardware with equal nominal budgets and are directly rank-comparable. Two round-1 *add-on* runs are reported separately and are not rank-comparable with the canonical tier: Opus 4.8-ultra used a max-effort plus workflow harness variant (the driver passed a maximum-effort flag and appended a standing directive to spawn parallel sub-agent workflows building subsystems —

Table 1: The seven runs. The canonical three (top) are rank-comparable: default harness, identical Hetzner cpx31 boxes (4 vCPU, 8 GB RAM, Ubuntu 24.04), equal nominal budget of 8 h plus a 1 h wrap-up. The Fable 5 row (middle) is a canonical-protocol addendum, run after the v1 draft under the identical protocol. Add-ons (bottom) are reported separately. Incident-driven reruns and resumes are documented in Appendix B.

Run	Tier	Driver CLI	Harness	Hardware	Budget
Opus 4.8	canonical	`claude -p` loop	default	cpx31	8 h + 1 h[a]
Opus 4.7	canonical	`claude -p` loop	default	cpx31	8 h + 1 h[a]
GPT-5.5	canonical	`codex exec` loop	default	cpx31	8 h + 1 h
Fable 5	addendum	`claude -p` loop	default	cpx31	8 h + 1 h[b]
Opus 4.8-ultra	add-on	`claude -p` loop	max-effort + workflow	cpx31	8 h + 1 h
Fable 5-ultra	add-on	`claude -p` loop	max-effort + workflow	cpx31	8 h + 1 h
Gemini 3.5	add-on	`agy` loop	default	operator macOS	incomplete[c]

[a] Original runs were degraded by an operator-side API-credit exhaustion about 48 minutes in; clean full-budget reruns were granted and the degraded artifacts preserved. [b] Its 1 h wrap-up overran to roughly 2 h because the non-ultra wrap script lacked a per-turn timeout; the operator cut it (Appendix B, incident 9). [c] Run parked early after API quota exhaustion followed by an credential expiry that could not be refreshed headlessly; also ran on the operator's Mac (the only practical `agy` host) and was scored there without the sandbox — a documented deviation.

lexer, parser, interpreter, property descriptors, builtins, RegExp, TypedArrays, Promise/async — then integrate and re-measure), and Gemini 3.5 Flash ran on different hardware with an incomplete budget. Table 1 summarizes the seven runs.

After the v1 draft of this report was complete, Claude Fable 5 — a newer model generation than the round-1 contestants — was evaluated in a two-run addendum under the identical protocol: the same Hetzner cpx31 hardware, the same 8 h + 1 h budget, the same scorer at the same pins, and the fixed harness (including the ASan-probe fix of Appendix B, incident 7), so no re-scoring of the earlier runs was required. The default-harness Fable 5 run is reported as a *canonical-protocol addendum*: it is protocol-identical to the canonical three but was not part of the simultaneous round-1 cohort, so the original canonical tables and figures are kept intact for the controlled comparison and the addendum rows are marked explicitly throughout. The Fable 5-ultra run used the same max-effort plus workflow variant as Opus 4.8-ultra and is reported with the add-ons.

All Claude-family runs (Opus 4.8, Opus 4.7, Fable 5, and the two ultras) were driven through the Claude Code CLI [1] in headless print mode; GPT-5.5 was driven through `codex exec`, and Gemini 3.5 Flash through `agy`. The driver CLI is part of each contestant's harness rather than a controlled variable: GPT-5.5's agentic scaffold (codex) and Gemini's (agy) differ from Claude Code in tooling and session management, so cross-vendor comparisons compare model-plus-scaffold stacks, not models in isolation (Section 7).

3.3 Infrastructure

Each contestant ran alone on a dedicated Hetzner cpx31 box (4 vCPU, 8 GB RAM, Ubuntu 24.04) provisioned with `clang` (the default compiler), `gcc`, `make`, `git`, and `python3`, a clone of the contest repository, and test262 pre-fetched at the pinned SHA. The reference runner (`runner/run262.py`) was available for self-measurement during development, but official scoring was performed only by the judge.

The agent ran headless inside tmux as an unprivileged `contestant` user under a driver loop (`scripts/launch-contestant.sh`): while wall-clock budget remained, the loop invoked the CLI

with the full brief on the first turn and a continuation prompt thereafter, with each turn capped by a 2,700 s timeout (added after a hung-turn incident; Appendix B). For Claude contestants the loop ran in FRESH mode: every turn starts a brand-new session that re-orients entirely from disk — re-reading `BRIEF.md`, `STATUS.md`, the existing source, and git history — rather than resuming a transcript. This mode was adopted after Claude Code's 32 MB `--continue` resume limit wedged a long-running session; with FRESH mode the transcript never grows across turns and all working state lives in the repository. The codex driver instead resumed the prior session each turn (`codex exec resume --last`). The continuation prompt explicitly instructed contestants not to start over if an engine already existed, and to self-measure with small samples and low parallelism to avoid exhausting memory.

Two pieces of operational hardening were added during the contest after contestants' own unthrottled test-suite runs ballooned to roughly 7 GB of resident memory and OOM'd their boxes: 4 GB of swap, and a periodic reaper that killed stray engine processes by age and resident-set size. At the end of the 8-hour budget each contestant received a 1-hour wrap-up turn whose prompt reminded it that the entire suite is scored as two buckets — negative tests and positive tests — before the final commit.

Scoring was never performed on contestant boxes. Submissions were collected and scored on separate, clean judge boxes with network lockdown; the scoring pipeline, sandboxing, and anti-gaming measures are described in Section 4.

4 Scoring Methodology and Measurement Integrity

4.1 Corpus and metrics

The scored corpus was derived from the `test/language` and `test/built-ins` trees of test262 [7] at the pinned commit `4249661388e5d3f92a85186213da140a6481490f`. Each test file was expanded into its strict, sloppy, module, and raw variants according to the rules of `INTERPRETING.md` [6], yielding a manifest of 91,280 scored cases (SHA-256 `74d59c3a2b078c585a711dba2e11e2b6 0864a616e28a2bdb9a624557c242ac60`). The `intl402/` (ECMA-402/ICU), `annexB/` (legacy web compatibility), and `staging/` (unratified) trees were excluded entirely; the two in-scope `CanBlockIsFalse` files were skipped with a recorded reason and removed from the denominator. The same denominator applied to every contestant.

Submissions had to honor a fixed command-line contract: `./js262 [--module] f1.js .. fN.js`, with the files evaluated in order in a single realm, the last file being the test itself and the preceding files the staged harness. A `print()` builtin was required; the `$262` host object was optional, but the 1,305 cases (across 664 files) that depend on it remained in everyone's denominator, so implementing it was a legitimate way to earn points (Section 3).

Four metrics were computed per submission. Each engine was built twice from the same source: an optimized `-O2` binary and an AddressSanitizer+UndefinedBehaviorSanitizer binary [10].

- **S1**: cases passed by the `-O2` binary.
- **S2** (*primary, ranked metric*): cases passed by the sanitizer binary, run with a 3× timeout multiplier and the address-space limit lifted to accommodate shadow memory. A pass under S2 therefore certifies both the observable test outcome and the absence of detected memory or undefined-behavior errors on that case.
- **S3**: a structured diff of the two runs (O2-only passes, ASan-only passes, sanitizer crashes, sanitizer timeouts), reported for transparency but not ranked.

- **S4**: wall-clock time of the O2 run, used only as the final tiebreaker (the brief ranked S2, then S1, then S4; no ties occurred).

Pass criteria distinguished two buckets. A *positive* case passed if the engine exited with status 0, no uncaught error, and (for non-async, non-raw cases) printed its per-case completion sentinel (Section 4.2). A *negative* case passed if the engine exited nonzero without being killed by a signal and the first non-empty line on `stderr` began with the expected error constructor name (e.g. `SyntaxError`, `TypeError`) at a word boundary. Termination by signal was always a failure, in both buckets. Async cases were judged at process exit: the runner waited for the engine to exit (or time out) and then required the per-case async completion token on `stdout`. The per-case timeout was 3 seconds; cases that timed out were retried once at 10 seconds, and the sanitizer run received a 3× multiplier (9 s/30 s). All official scoring ran sequentially on a clean judge box, never on contestant machines, with one documented exception: the incomplete Gemini 3.5 Flash run was scored on the operator's Mac with four parallel jobs and without the sandbox (Appendix B).

4.2 Anti-gaming hardening

The runner source was present on every contestant's box, so the staging pipeline was designed under the assumption that a contestant could read it. The threat model distinguishes lazy or structural shortcuts, which the runner defeats outright, from determined in-band token grepping, which is instead made detectable.

Completion sentinel. Before this control was added, a do-nothing stub that simply exited 0 scored approximately 79% of the corpus — essentially every positive non-async case, where a clean exit alone sufficed to pass. The runner therefore appends `print("Test262Sentinel:<token>")` with a random per-case token to every staged non-raw test; positive non-async cases must emit their token to pass. The same stub scores 0 under the hardened runner.

Uniform trailer. The sentinel trailer is appended to *all* non-raw staged tests — positive, negative, and async alike — so the presence or absence of a trailer reveals nothing about the expected outcome. This removes the structural classification oracle: an engine that greps tokens out of the staged source cannot tell positives from negatives, so no single trivial strategy passes both buckets.

Frontmatter stripping. The test262 metadata block, which declares the expected outcome of negative tests, was stripped from every staged non-raw test and replaced by an equal number of blank lines (preserving line numbers for error reporting).

Opaque filenames. Harness files (`assert.js`, `sta.js`, the async handler, includes) were staged under randomized names of the form `h_<i>_<rand>.js`, and test filenames were likewise randomized, so `argv` never revealed whether a case was async or which includes were present.

Per-case async tokens. The async completion handler was a runner-controlled copy of `doneprintHandle.js` with the public `Test262:AsyncTestComplete/...Failure` markers replaced by per-case random tokens, so the fixed public strings could not simply be echoed.

Secret computed-output probes. Each official scoring run generated a fresh set of 40 secret probes with random operands whose answers were computed judge-side; the probe file was never committed and never placed on a contestant box. Each probe required the engine to print `R=<computed value>`, so an engine that echoes staged tokens without executing JavaScript fails every probe. The probes were run against *both* the O2 and sanitizer binaries, and the scoreboard raised a cheat flag whenever a binary scored high on the public suite but low or missing on its probes. In the official results all engines scored 40/40 on both binaries, with one exception: GPT-5.5 scored 37/40 on both binaries, reflecting genuine computed-output errors rather than gaming

(corroborated by the observational speed benchmarks, Section 5). One scoring-harness bug initially false-tripped the cheat flag for every engine (the probe pass ran the sanitizer binary without sanitizer-mode limits); it was diagnosed, fixed, and all engines were re-scored with identical S2 results (Appendix B).

Any marker placed in the staged source can in principle be grepped and echoed without execution; no in-band scheme prevents that. The layered defense is: the uniform trailer removes the classification oracle, the secret probes make a high public score without real execution detectable, and a transcript and binary audit serves as the operator's backstop.

4.3 Sandboxing and verification

Every engine invocation during scoring ran inside a bubblewrap sandbox [4] configured with `--unshare-all` (no network), a fresh PID namespace, and an unprivileged uid (65534); per-case temporary directories were read-only to the sandboxed engine, and neither the judge's home directory nor the test262 tree was mounted. Resource limits were `RLIMIT_AS` of 2 GiB (lifted for the sanitizer binary, whose shadow memory requires a large address-space reservation) and a 1 MiB cap on each output stream.

Before any case was run, the build was verified: `readelf` confirmed that the sanitizer binary actually linked the sanitizer runtimes (at least 50 `__asan_` and 20 `__ubsan_` symbols, or the corresponding dynamic libraries) — a submission whose "ASan build" silently omitted instrumentation would have its S2 reported as zero — and an `ldd` audit confirmed the binaries depended only on libc, libm, and the sanitizer runtime, ruling out a linked third-party engine. Each scoreboard recorded the manifest SHA-256, the pinned test262 commit, the engine binary SHA-256s, and the scoring-box environment. Scoring boxes were separate, freshly provisioned machines placed under a network lockdown (verified by an egress probe) before the submission was cloned, so the engine could not fetch anything at score time.

4.4 Documented deviations

The score is not a claim of standards-grade test262 conformance. The runner followed `INTERPRETING.md` where practical; the deliberate deviations, each applied identically to every contestant, were:

1. **Negative-test phase not enforced.** Only the error type was checked, not the `parse`/`resolution`/`runtime` phase; at the pinned commit, 4,581 of the 4,647 in-scope negative tests are `parse`+`SyntaxError`. The phase was recorded for analysis.

2. **`$262` optional.** Tests requiring the `$262` host object simply fail for engines that omit it, but remain in the common denominator.

3. **Raw tests keep frontmatter.** The 29 in-scope `raw` files must be evaluated byte-for-byte unmodified, so they were exempt from metadata stripping; the theoretical leak is limited to those files.

4. **`CanBlockIsFalse` skipped.** The contract host is a plain shell process that can block; the two such files were removed from the denominator.

5. **Async judged at process exit.** An engine that prints the completion token but never exits is a timeout failure.

6. **Module harness delivery.** Harness files were passed as separate `argv` scripts evaluated before the module test, not concatenated into it; module fixtures were staged as sibling copies in the temporary directory.

7. **Scope exclusions.** `intl402/`, `annexB/`, and `staging/` were excluded.

8. **Anti-gaming staging.** The sentinel trailer, frontmatter stripping, opaque filenames, and token randomization of Section 4.2 are themselves departures from vanilla staging.

9. **Timeout and resource policy.** The 3 s/10 s-retry timeouts, address-space limit, and output caps are contest policy, published in the brief; test262 prescribes none of them.

4.5 Runner validation

The runner was validated along four lines before any official score was accepted.

Unit suite and adversarial stubs. A 54-test unit suite covered manifest expansion and skip rules, staging (frontmatter stripping, harness ordering, opaque names, the uniform trailer across all three case classes, per-case async tokens, module fixtures), the full outcome-classification table, sentinel and probe logic, timeout and process-group kill handling, and the incomplete-run denominator (unrun cases are materialized as failures, so a crashed scoring run cannot inflate a percentage). Adversarial stub engines exercised the hardening directly: a structural cheat that greps staged tokens passed positives but no non-raw negative; a frontmatter-grep cheat scored zero on non-raw negatives; sentinel-echo and structural cheats passed zero probes while a real engine (Node.js) passed all of them; and the do-nothing `exit 0` stub scored zero.

Oracle A: full-corpus known engine. QuickJS-NG 0.15.0 [9] run under this runner over the full manifest scored 78,710/91,280 (86.23%) with zero runner errors, within 0.08% of its pre-hardening run — evidence that the anti-gaming staging does not perturb a real engine's results. Bellard QuickJS 2025-09-13 [2] scored 82.47% with the same result shape.

Oracle B: case-by-case official-harness diff. The same engine (Node.js v20.19.3) was run on representative shards (modules, hashbang, global code, and several built-ins trees) under both the official `test262-harness` and this runner. Of 1,855 jointly run cases, every verdict disagreement was individually explained and attributable to the official harness's enumeration quirks, eshost module handling, the `$262` shim, or engine-level error-type differences — unexplained runner discrepancies: zero. This validates the runner's interpretation of test262 metadata on those shards, though it is not a whole-corpus same-engine comparison.

End-to-end pipeline. The complete `score.sh` pipeline was exercised against a stub engine on a fresh Ubuntu 24.04 box, confirming the network lockdown (egress-probe-verified), unprivileged build, sanitizer linkage verification, in-sandbox execution, probe behavior, and scoreboard provenance fields end to end.

5 Results

All scores below were produced by the v2-verified scoring pipeline against the pinned test262 denominator of 91,280 cases (SHA `4249661388e5d3f92a85186213da140a6481490f`). Cheat flags were clear for every run, and the 40 secret computed-output probes passed 40/40 on both the O2 and sanitizer binaries for every contestant, with one exception noted below (GPT-5.5, 37/40 on both binaries). The canonical three ran under identical conditions; the Fable 5 row is the canonical-protocol addendum, run after the v1 draft under the identical protocol and pins (Section 3); the

Table 2: Conformance results over 91,280 test262 cases. S2 (ASan-clean passes) is the primary metric; S1 counts passes of the O2 binary. "San.-only" counts cases that failed only on the sanitizer-instrumented binary. Canonical three and the Fable 5 addendum above the rule; add-ons below are not rank-comparable.

Contestant	Tier	S2	S2 (%)	S1 (%)	San.-only	Wall (s)
Fable 5‡	addendum	89,191	97.71	97.72	6	782.4
Opus 4.8	canonical	81,160	88.91	88.95	38	648.4
Opus 4.7	canonical	59,439	65.12	65.30	281	702.8
GPT-5.5	canonical	41,702	45.69	45.99	856	1,383.8
Fable 5-ultra	add-on	88,347	96.79	96.87	84	697.9
Opus 4.8-ultra	add-on	84,740	92.84	92.93	246	875.4
Gemini 3.5	add-on	29,549	32.37	32.72	–	516.9†

‡Canonical-protocol addendum: identical protocol, hardware, and pins as the canonical three, but run after the v1 draft and not part of the simultaneous round-1 cohort (Section 3). †Gemini 3.5's run was incomplete and was scored on a Mac rather than a Hetzner cpx31 box, with four parallel jobs (Appendix B); its wall-clock time is not comparable to the other rows, and the unsandboxed Mac scoring path did not produce the structured S1/S2 diff (San.-only).

add-on rows (including Fable 5-ultra) are reported separately and are not rank-comparable with the canonical tier.

5.1 Conformance

Table 2 reports the primary metric S2 (ASan-clean passes), the secondary metric S1 (O2 passes), sanitizer-only failures, and the O2 run's scoring wall-clock time (S4); Figure 1 plots the pass rates.

Claude Opus 4.8 won the canonical tier with 81,160 ASan-clean passes (88.91%). Claude Opus 4.7 reached 65.12% and GPT-5.5 45.69%, a spread of 43.2 percentage points between the best and worst canonical runs; the two Opus generations alone were separated by 23.8 points. Among the five round-1 runs, the highest score belonged to Opus 4.8-ultra, the max-effort-plus-workflow variant, at 84,740 (92.84%) — 3.9 points above the default Opus 4.8 run. For reference, QuickJS-NG 0.15.0 scored 86.23% under the same runner (Section 4), placing four of the seven model-built engines above an established production interpreter on this denominator.

The overall champion of the study came from the addendum. Fable 5 scored 89,191 ASan-clean passes (97.71%) under the *default* harness — 4.87 points above the prior best (Opus 4.8-ultra at 92.84%) and 11.5 points above QuickJS-NG's 86.23% under the identical runner and denominator. Only 2,089 of the 91,280 cases failed, of which 224 are `$262.agent` (multi-threaded agent) cases that no contestant was expected to attempt. The engine implemented SharedArrayBuffer and Atomics, a full ES module system with top-level await (built on fibers), BigInt, explicit resource management (`using`/`await using`), and iterator helpers; its final commits were Date-setter and RegExp group-name spec minutiae. Its wrap-up self-measurement of 492/500 (98.4%) overestimated the official score by roughly 0.7 points. Fable 5-ultra, the workflow variant of the same model, scored 88,347 (96.79%) — 0.92 points *below* its own default-harness counterpart (Section 6).

The gap between S1 and S2 acts as a memory-safety quality signal: a large gap means the engine passes tests whose executions trigger AddressSanitizer or UBSan findings [10]. The top engines showed small sanitizer-only failure counts — 6 for Fable 5 (the lowest of any run) and 38 for Opus 4.8 — while Opus 4.7 recorded 281 and GPT-5.5 856. Fable 5-ultra recorded 84; the Opus 4.8-ultra tree, despite the highest round-1 score, accumulated 246 sanitizer-only failures. GPT-5.5

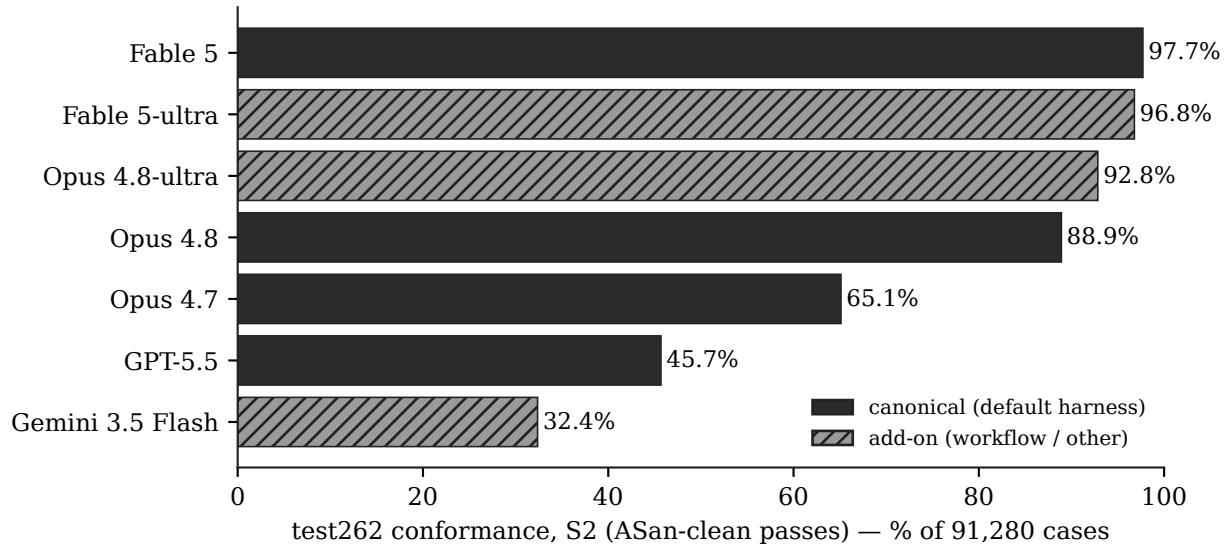

Figure 1: S2 pass rates over the 91,280-case denominator, sorted by score. Solid bars are canonical-protocol runs (the canonical three plus the Fable 5 addendum); hatched bars are add-on runs (separate tier, not rank-comparable).

was also the only contestant to miss secret probes: 37/40 on both the O2 and sanitizer binaries, all genuine computed-output mismatches rather than harness artifacts (the earlier all-engine 0/40 ASan probe trip was a diagnosed harness bug; see Appendix B).

5.2 Code shape and completeness

Table 3 summarizes the submitted codebases after source audit; Figure 2 relates these effort proxies to score.

Architecture split cleanly along the score ranking: every run above 80% shipped a modular multi-file engine (27–60 source files), while the three lowest-scoring runs (Opus 4.7, GPT-5.5, Gemini 3.5) each produced a monolith — Opus 4.7's engine lived in 2 files and GPT-5.5's in a single file of 10,777 lines. We report this as a correlation, not a causal claim; with n=7 runs we cannot separate architectural choice from underlying model capability.

Completeness differed in which feature blocks engines attempted. The largest single block was the Temporal proposal, roughly 9,206 cases of the denominator: engines attempted varying subsets of it (Opus 4.8-ultra, for example, integrated a partial Temporal implementation via a parallel agent late in its run, per its commit log).

Commit count was a misleading effort proxy. GPT-5.5 made the most commits of any contestant (282) and scored lowest in the canonical tier, while Opus 4.8-ultra submitted just 60 commits and posted the highest round-1 score at 92.84%; Fable 5 made the fewest commits of any run (37, over just 9 long driver iterations) and scored highest in the study. Audited LOC ordered the engines more sensibly: the four largest codebases (19,901–43,599 LOC) were the four highest-scoring runs.

5.3 Observational speed benchmark

We ran six checksum-validated workloads (`fib`, `json`, `mathloop`, `objchurn`, `sort`, `strings`) against each submitted O2 binary, taking the median of 5 runs; a time counts only if the printed checksum

Table 3: Code shape of the submitted engines. LOC and file counts are from post-audit measurements of the submitted source tree (worktree-inflated naive counts excluded; see Section 5, parallel sub-agent usage). Gemini 3.5's tree was not audited (incomplete run).

Contestant	Commits	LOC	Files	Driver iters	Architecture
Fable 5‡	37	30,043	27	9	modular
Opus 4.8	103	19,901	31	215	modular
Opus 4.7	149	15,683	2	744	monolith
GPT-5.5	282	10,777	1	44	monolith
Fable 5-ultra	81	43,599	60	–	modular
Opus 4.8-ultra	60	33,221	43	752	modular
Gemini 3.5	139	–	–	2,328	monolith

‡Canonical-protocol addendum (Section 3). A driver-iteration count was not extracted for Fable 5-ultra.

Table 4: Observational speed benchmark: median of 5 runs, milliseconds, on the contest Hetzner boxes (Gemini 3.5 on a Mac; times not comparable). FAIL = engine ran but printed an incorrect checksum (wrong computed output). Opus 4.7 was not included in the verified benchmark data.

Workload	Opus 4.8	Fable 5	Opus 4.8-ultra	Fable 5-ultra	GPT-5.5	Gemini 3.5†
fib	4,285	1,907	2,838	2,430	FAIL	FAIL
json	107	51	59	69	FAIL	57
mathloop	275	584	282	649	FAIL	FAIL
objchurn	250	403	259	434	FAIL	505
sort	634	339	299	290	FAIL	FAIL
strings	191	231	172	223	FAIL	FAIL

†Different hardware (operator's Mac); absolute times not comparable to other columns; run incomplete (Appendix B).

is correct. This benchmark was observational only — it was never a contest metric and contestants were not asked to optimize for it. Table 4 reports the results.

The headline finding here is about correctness, not speed. GPT-5.5's engine produced wrong output on *all six* workloads: it built cleanly, ran to completion, and printed plausible-looking but incorrect results. This is silent mis-execution — the failure mode in which an engine neither crashes nor throws but computes wrong answers — and it corroborates the same engine's 37/40 score on the secret randomized probes. An engine can pass 45.69% of test262, much of it negative (syntax-error) tests and simple positive assertions, while still mis-executing ordinary numeric and string workloads. Gemini 3.5's incomplete engine produced wrong output on 4 of 6 workloads (only `json` and `objchurn` validated). All four Claude-built engines produced correct checksums on all six workloads. Among them, Fable 5 posted the fastest `fib` (1,907 ms) and `json` (51 ms) times of any engine in the study; Opus 4.8 won `mathloop` (275 ms) and `objchurn` (250 ms), Fable 5-ultra won `sort` (290 ms), and Opus 4.8-ultra won `strings` (172 ms).

5.4 Token usage

Token accounting comes from the Claude session JSONL logs and is therefore complete only for the five Claude runs. For GPT-5.5, codex accounting is best-effort cumulative-per-session (approximately 0.75M output tokens observed) and is not comparable; the agy harness used for Gemini 3.5 logs no token counts at all. Provider-dashboard reconciliation is future work. Table 5 reports

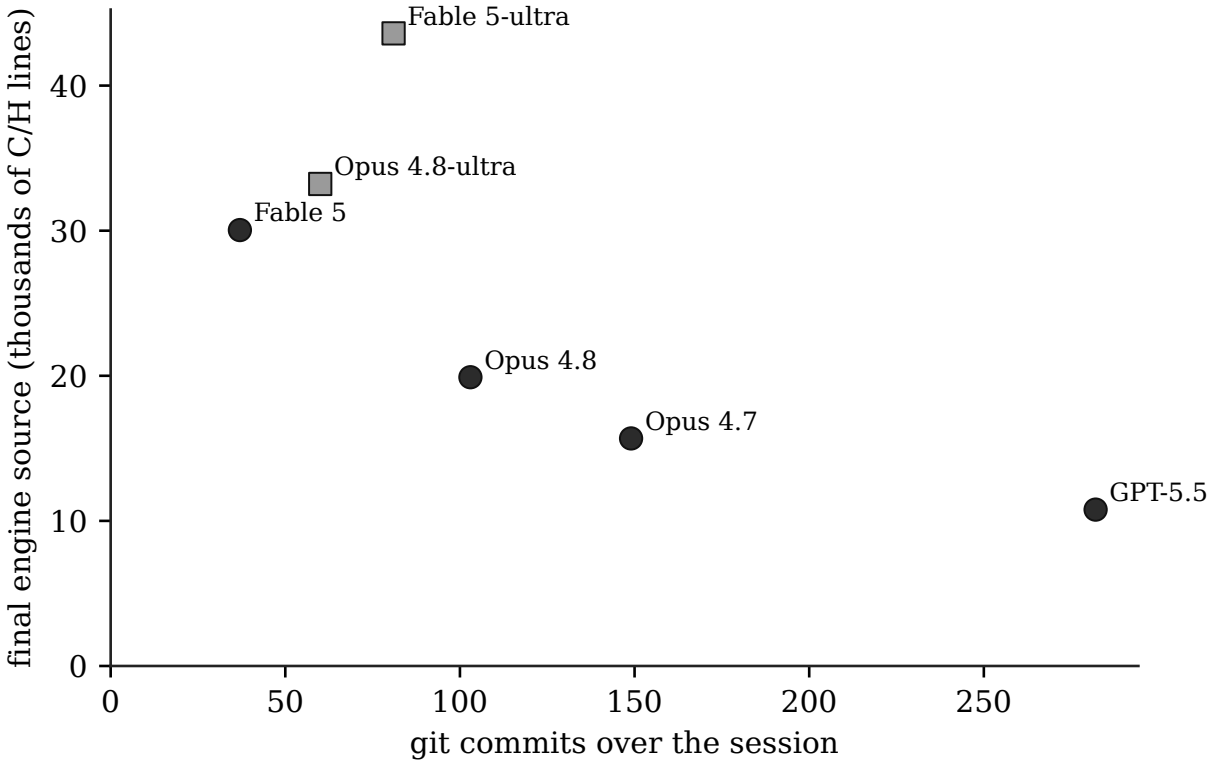

Figure 2: Git commits versus final audited LOC per run. Commit count is a poor predictor of codebase size; audited LOC and novel tokens (Figure 3) track score and codebase size more closely.

the Claude runs; Figure 3 plots novel tokens against final codebase size.

Four observations stand out. First, token-efficiency differences between models were large: Fable 5 produced roughly 6.2K lines of final audited code per million output tokens, versus roughly 0.66K for Opus 4.8 — an order-of-magnitude gap in output tokens spent per surviving line. Second, novel-token volume (cache-creation input) tracked final codebase size far better than commit count did across runs. Third, the default Opus 4.8 run spent about 4x the output tokens of its own workflow variant (30.34M vs. 7.51M) yet scored lower (88.91% vs. 92.84%); Opus 4.8-ultra reached the highest round-1 score on a small fraction of the default Opus 4.8 run's output tokens. Fourth, the addendum set the efficiency record of the study: Fable 5's 10.97M novel tokens are the lowest of any Claude run — roughly one-twelfth of the default Opus 4.8 run's 136.88M — yet it produced the highest score, roughly 13.7x Opus 4.8's score per novel token. Fable 5-ultra spent roughly 3.4x Fable 5's billable tokens (55.8M vs. 16.2M) for a score 0.92 points lower.

5.5 Parallel sub-agent usage

The round-1 workflow variant made heavy, verifiable use of parallel sub-agents via git worktrees. Opus 4.8-ultra accumulated 14 or more agent and workflow worktrees over its run. It submitted exactly 60 commits — versus 103–282 for the round-1 default runs — reflecting fewer, larger integration commits that merged sub-agent work back into the mainline; its log contains explicit "integrate …(parallel agent)" commits for RegExp, String/Math, and Temporal work. Its high driver-iteration count (752) combined with low session output tokens (7.51M) is consistent with an orchestrate-and-integrate loop rather than single-threaded generation.

Table 5: Token usage for the five Claude runs, from session JSONL logs. "Novel" is cache-creation input (text entering the context for the first time); billable approximates input + output + cache-creation. GPT-5.5 and Gemini 3.5 are omitted: their harnesses do not provide comparable accounting. Assistant-message counts were not extracted for the two addendum runs.

Contestant	Asst. msgs	Output	Novel	Cache read	Billable
Fable 5‡	–	4.87M	10.97M	558M	16.2M
Opus 4.8	16,105	30.34M	136.88M	2.99B	169.1M
Opus 4.7	13,493	6.36M	17.99M	4.83B	24.4M
Fable 5-ultra	–	11.73M	42.96M	703M	55.8M
Opus 4.8-ultra	11,832	7.51M	33.24M	1.82B	41.7M

‡Canonical-protocol addendum (Section 3).

All shape numbers in Table 3 are post-audit: worktree checkouts inflate naive repository line counts well beyond the engine source proper, so we measured the audited `src/` trees only. The source audit of the Opus 4.8-ultra tree found no third-party engine code; its 3,034-line Unicode table was generated from the Unicode Character Database via a Python script, disclosed in its commit message and permitted by the brief.

The addendum's workflow run, Fable 5-ultra, used the parallelism differently: a stub-scaffold strategy. It scaffolded the entire engine early behind stub feature flags, had parallel workflow agents land whole modules mid-run — its log contains commits such as "interp_expr.c + interp_stmt.c land (workflow agents)" — and dropped the stub flags one by one as the modules became real, closing with a late integration surge of 18 commits in its final hour. The result was the largest engine of the study: 43,599 audited lines across 60 files.

Commit subjects also record contestants' own progress estimates (`results/metrics/*.gitlog`). These are sampled *self*-measurements on the contestants' own harnesses, distinct from official scores, but they landed close: Opus 4.7's `e1c0c65` claimed "62%+ pass rate" (official: 65.12%); Gemini 3.5's `6245fc5` recorded a self-measured 32.20% (official: 32.37%); and Opus 4.8-ultra's last status commit estimated ~89.2% on samples, under-predicting its official 92.84%. In the addendum, Fable 5's wrap-up self-measurement of 492/500 (98.4%) overstated its official 97.71% by roughly 0.7 points. None of the self-estimates materially overstated the official result; the largest overstatement was Fable 5's, at roughly 0.7 percentage points on a 500-case sample.

6 Analysis and Discussion

Capability ordering at equal harness. Among the canonical three, the S2 ordering was unambiguous: Claude Opus 4.8 (88.91%) > Claude Opus 4.7 (65.12%) > GPT-5.5 (45.69%) — a spread of roughly 43 percentage points under identical hardware, budgets, prompts, and scoring (Table 2). Several independent signals indicate that this spread reflects real semantic capability rather than harness noise. The secret computed-output probes were 40/40 for both Opus runs but 37/40 (on both binaries) for GPT-5.5, with the misses being genuine wrong computations. The observational speed bench corroborates this: Opus 4.8 produced checksum-correct output on all six workloads, while GPT-5.5 produced wrong output on all six. Sanitizer-clean failure counts follow the same gradient (38 ASan-failing cases for Opus 4.8 versus 281 for Opus 4.7 and 856 for GPT-5.5), as does source structure: Opus 4.8 built a modular multi-file engine, while Opus 4.7 and GPT-5.5 produced monoliths of 2 and 1 source files respectively (Table 3).

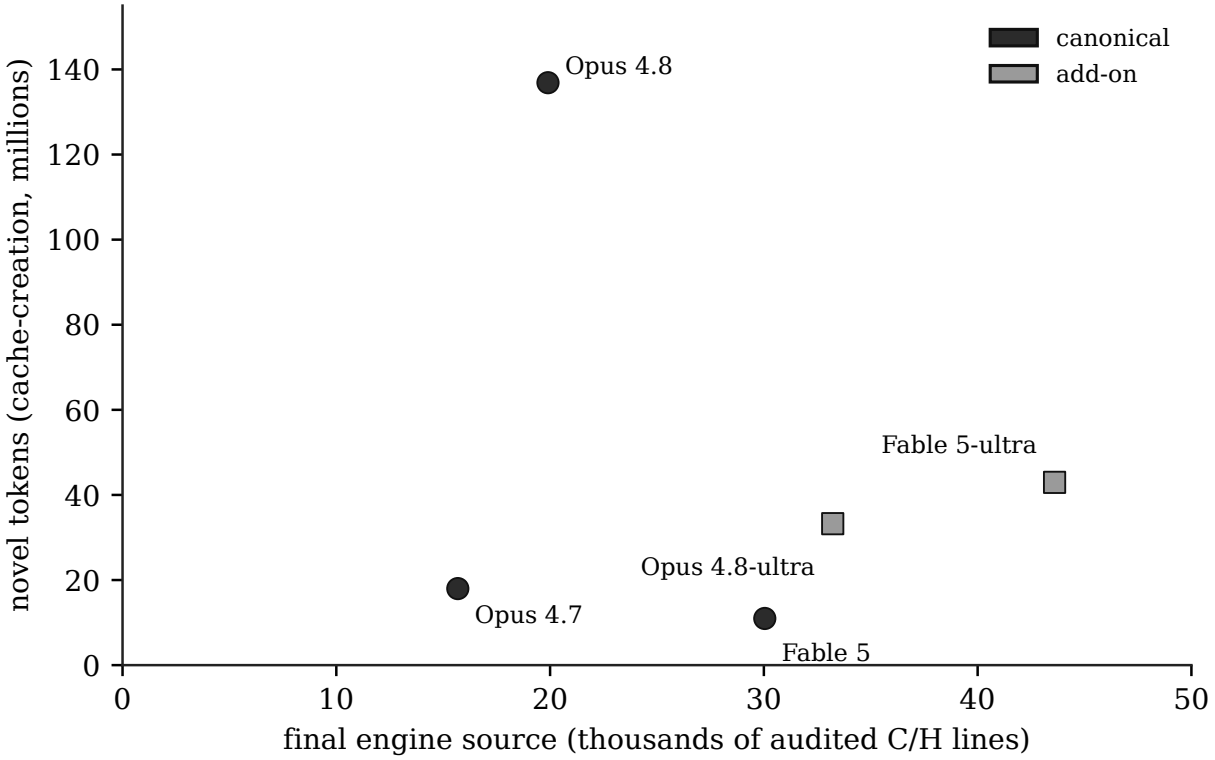

Figure 3: Novel (cache-creation) tokens versus audited final LOC. Novel-token volume tracks final codebase size far better than commit count does.

The harness effect is model-dependent. The most interesting result of the contest is not the ranking but the interaction between model and harness. With the addendum, two models ran under both the default harness and the max-effort-plus-workflow variant, and the workflow helped exactly one of the two. For Opus 4.8 it added +3.93 points (88.91% to 92.84% S2), making Opus 4.8-ultra the strongest round-1 engine. For Fable 5 it subtracted 0.92 points (97.71% to 96.79%): the default-harness Fable 5 run beat its own orchestrated variant while spending roughly 3.4x fewer billable tokens (16.2M versus 55.8M). What the two pairs establish is that orchestration scaffolding is not a free or uniform improvement: the same workflow yielded a clear gain for one model and a loss for the other. Both ultra runs did exploit the workflow's parallelism (named git worktrees, parallel module landings, large integration commits), so the divergence appears to lie in how each model used the structure, not in whether it engaged with it. Controlled study of harness–model interaction is warranted before treating agentic workflow layers as generally beneficial.

Token volume is not quality. The default Opus 4.8 run emitted 30.34M output tokens; Opus 4.8-ultra emitted 7.51M — roughly a quarter — and scored 3.93 points higher (Table 5), though the comparison is confounded by the max-effort setting. Within our accounting, novel tokens (cache-creation) track final codebase size far better than commit count does. The contrast in generation style between the two strongest default-harness models is stark: Fable 5 produced about 6.2K lines of final audited source per million output tokens against about 0.66K for Opus 4.8. The two working styles behind those figures are equally far apart: Fable 5 worked in 9 long, decisive driver iterations, shipping 37 commits on 4.87M output tokens, while Opus 4.8 micro-iterated through 215 driver iterations and 16,105 assistant messages on 30.34M output tokens (Figure 2). The more

verbose style was not the more efficient one: Fable 5 reached the study's highest score (97.71%) on the lowest novel-token volume of any Claude run (10.97M), roughly 13.7 times Opus 4.8's score per novel token (Section 5).

Failure modes differ qualitatively. The score gap understates a qualitative difference in *how* engines fail. GPT-5.5's engine paired a permissive parser with silent mis-execution: it ran most inputs to completion but computed incorrect results, as shown by its 37/40 probe result and wrong checksums on all six bench workloads. This profile — high apparent progress with quietly wrong computation — is more concerning for any downstream use than an engine that honestly lacks features and fails loudly, because the errors are invisible without an external oracle. A conformance score alone does not distinguish these regimes; the probes and the checksum-validated bench were essential to detecting it.

What eight hours buys. The headline of the study is now the addendum's: Fable 5's default-harness run reached 97.71% against QuickJS-NG 0.15.0's 86.23% under the identical runner and denominator — 11.5 points above the production reference. Opus 4.8 (88.91%), Opus 4.8-ultra (92.84%), and Fable 5-ultra (96.79%) also exceeded that figure; every Claude-built engine exceeded what the operators expected a single budgeted session to produce. This comparison must be read narrowly. QuickJS-NG [9] is a complete, memory-safe, production-deployed engine developed over years, with goals — footprint, speed, embeddability, full-spec coverage including features outside this denominator — that a one-day pass-rate sprint does not share. Matching or exceeding its pass rate on this filtered slice of test262 is not a claim of production parity. The defensible statement is more modest and still notable: under this contest's rules, roughly eight hours of frontier-model effort produced from-scratch interpreters whose conformance on this denominator falls in the range of — and, for the newest model, well above — a mature production engine's.

7 Threats to Validity

We enumerate the principal threats to the validity of the comparisons reported here.

1. **Unequal effective wall-clock despite equal nominal budgets.** All canonical contestants received the same nominal budget (8 hours plus a 1-hour wrap-up), but several runs were disrupted by operator-side or harness-side incidents (Appendix B): API-credit exhaustion roughly 48 minutes into the original Claude Opus 4.7 and Claude Opus 4.8 runs, a session wedged by a 32 MB resume limit in the driver, overnight authentication failures during a logged-out window, and box-level memory pressure caused by contestants' own unthrottled debug processes. Every run materially degraded by operator or harness error received a clean rerun or resume, and the degraded artifacts were preserved rather than discarded. Nevertheless, effective productive wall-clock was not perfectly equal across contestants, and residual effects on relative scores cannot be ruled out.

2. **Single run per configuration.** Each contestant and harness configuration was executed once per round. Long agentic coding sessions plausibly exhibit substantial run-to-run variance; round 5's blind replication provides the series' first empirical bound — six unchanged configurations swung by up to −2.35x raw on identical workloads (Section 11.6) — but at one run per cell per round, finer orderings remain provisional. Score differences of a few percentage points between adjacent contestants may therefore not be stable under replication;

the largest gaps (for example, Claude Opus 4.8 at 88.91% S2 versus GPT-5.5 at 45.69%) are unlikely to be variance artifacts.

3. **Gemini 3.5 Flash is not comparable.** The Gemini 3.5 Flash run deviated from the protocol in three ways: it ran on the operator's Mac rather than a Hetzner box (the only practical host for the `agy` harness), it ended early after quota exhaustion followed by an credential expiry that could not be refreshed headlessly, and its engine was scored on the Mac without the bubblewrap sandbox. Its score (32.37% S2) is reported for completeness only and supports no cross-model conclusion.

4. **Self-measurement versus official score.** Contestants measured their own progress against locally staged tests on their own boxes, while official scores were computed judge-side on separate clean machines against the anti-gaming staged suite (Section 4), with its timeout, sandbox, and sanitizer requirements. A contestant's internal pass-rate estimates need not match its official score, and a contestant optimizing against its own measurements may have allocated effort differently than the official metric rewards. Only judge-side scores are reported.

5. **Operator-in-the-loop incident handling.** Incidents were diagnosed and remediated manually during the contest (rerun grants, driver fixes, swap and process-reaper additions, a scoring-harness bug fix followed by a full re-score). Such judgment calls could in principle bias outcomes. Mitigations were a single fairness rule applied uniformly—any run materially degraded by operator or harness error received a clean rerun or resume—and preservation of all degraded artifacts for inspection. The handling was not, however, blind to contestant identity.

6. **Token accounting is not comparable across vendors.** Claude-family token counts come from session JSONL logs with per-message granularity. The codex accounting for GPT-5.5 is a best-effort cumulative-per-session figure, and the `agy` harness used for Gemini 3.5 Flash logs no token counts at all. Cross-vendor efficiency comparisons are therefore not supported by the data in this report; reconciliation against provider billing dashboards is future work.

7. **Possible undetected gaming.** The scoring pipeline combined anti-gaming staging, secret randomized computed-output probes regenerated for every scoring run, per-case completion sentinels, sanitizer-linkage and dynamic-dependency verification, sandboxed execution, and source audits; all cheat flags were clear in the final verified scoring pass. These defenses raise the cost of gaming substantially but do not prove its absence. Transcript and source auditing was best-effort, not exhaustive.

8. **Benchmark composition.** test262 weights language areas by case count, not by importance. The Temporal proposal alone contributes approximately 9,206 cases (10.1% of the 91,280-case denominator), so a single decision to implement or skip one large area moves the headline score by several points. S2 percentages should be read as suite coverage under this particular weighting, not as a one-dimensional measure of engine quality.

9. **Addendum wrap-up overrun (Fable 5).** The non-ultra wrap-up script lacked the main driver's per-turn timeout, so a single turn spanning the one-hour deadline ran unbounded; Fable 5's wrap-up consequently overran to roughly 2 hours before the operator cut it (Appendix B, incident 9). Other contestants' wrap-ups ran roughly 1–1.5 hours under the same script, and one was operator-cut at 51 minutes, so the overrun was a harness gap rather than

a contestant action and the cut restored approximate parity. The final commit landed during the overrun window was spec-minutiae polish, so we judge the score effect to be small, but wrap-up time was not exactly equal across runs.

10. **The addendum is not contemporaneous with round 1.** The two Fable 5 runs were performed days after the round-1 cohort, once the v1 draft of this report was complete, under the identical protocol, hardware, pins, and fixed harness. They are protocol-comparable but were not part of the simultaneous round-1 cohort, and Fable 5 is a different (newer) model generation than the round-1 contestants. The canonical three-way comparison is therefore kept intact, and addendum rows are marked throughout; cross-cohort comparisons inherit whatever drift a few days and a model generation introduce.

8 Conclusion

Three frontier coding models—Claude Opus 4.8, Claude Opus 4.7, and GPT-5.5—each wrote a JavaScript engine in C from scratch in a single budgeted session (8 hours plus a 1-hour wrap-up) on identical Hetzner cpx31 hosts, and were scored offline against 91,280 test262 cases at a pinned SHA under a hardened, anti-gaming runner with ASan-clean passes (S2) as the primary metric. Claude Opus 4.8 led the canonical three at 88.91% (81,160 passes), followed by Claude Opus 4.7 at 65.12% and GPT-5.5 at 45.69%; for reference, the mature QuickJS-NG engine scored 86.23% under the same runner [9]. After the v1 draft was complete, Claude Fable 5—a newer model generation than the round-1 contestants—was evaluated in a two-run addendum under the identical protocol, hardware, and pins; its default-harness run scored 97.71% (89,191 passes)—the strongest greenfield build result in the study, 11.5 points above the QuickJS-NG reference—while emitting the fewest novel tokens of any greenfield Claude run.

The harness variants showed that the scaffolding wrapped around a model can shift outcomes in either direction, and helped exactly one of the two models that ran both. With the max-effort-plus-workflow harness, Opus 4.8-ultra reached 92.84% (84,740 passes)—the highest round-1 score and above the QuickJS-NG reference—while emitting roughly a quarter of the default Opus 4.8 run's output tokens (7.51M versus 30.34M). Fable 5-ultra scored 96.79%, 0.92 points below Fable 5's own default-harness run, at roughly 3.4x the billable tokens (55.8M versus 16.2M). The workflow add-on was therefore not uniform in its benefit, and add-on results are reported separately rather than rank-compared with the canonical tier.

The contest also functioned as a feasibility test for objective, hard-to-game, single-session engineering evaluations. Frontmatter stripping, uniform trailers, randomized opaque filenames, per-case completion sentinels, secret computed-output probes run against both binaries, sanitizer-linkage verification, and sandboxed judge-side execution together closed the gaming avenues we tested: a do-nothing exit-0 stub that scored approximately 79% of the corpus before sentinel hardening scored zero after, and the probes independently surfaced GPT-5.5's silent mis-execution (37/40 probes on both binaries) that its raw pass count alone would not reveal. The cost was a set of documented deviations from strict test262 semantics (Section 4), applied identically to every contestant.

The greenfield rounds were followed by three further contests on the same engines, reported as addenda in Sections 9, 10, and 11: a "last mile" round that restarted the field from the champion engine and raced it to fix the remaining failures under a full-suite no-regression gate, a "nitro" round that raced the same field to make that engine fast under the same gate, and a blind replication of the nitro round whose single intended change was a revised system prompt (an alternate variant of the harness's production system prompt) on the two Fable 5 runs.

Their headline findings extend rather than overturn the greenfield ones: the gate, not raw production, decided all three boards (five of eight runs zeroed in round 3, two of eight in round 4, four of eight in round 5 — with the Claude Opus 4.7 variants zeroed every time, leaving that family 0-for-6 on gates); the parallel-workflow harness took all three contest wins while remaining model-dependent; the engine itself was lifted from 97.71% to 99.07% of the suite and then made 4.79x and finally a gate-clean 5.19x faster; and the replication's six unchanged control runs swung round-over-round by as much as — and in one case roughly twice — the intended treatment's effect, bounding what any single-run comparison in the series can claim.

Future work includes reconciling token usage against provider billing dashboards to enable cross-vendor efficiency comparisons; extending the canonical tier to additional models; extending round 5's first variance measurement (six unchanged configurations re-run blind) to repeated sessions per configuration; and an unbounded "run to 100%" phase that continues from the same submitted engines without a time budget, to study where each engine saturates.

9 Round 3: The Last Mile — racing a shared champion engine to suite completion

The first round measured greenfield building: eight hours from an empty directory to an engine. The third round measures the opposite regime — finishing. Every contestant started from the *same* champion engine and raced to fix its remaining failures, under a hard rule that nothing already passing may break. In contest numbering, round 1 is the greenfield build of Sections 3–5, round 2 is the Fable 5 addendum that produced the champion, and this section reports round 3 ("the last mile"). All eight runs are officially scored: Wave A (six runs) ran first, and Wave B (two runs, workflow variants of the Opus models) followed the next day. Five of the eight were zeroed by the no-regression gate; only three runs survive onto the final board. Round 4, a speed race over the engine this round produced, is reported in Section 10.

9.1 Design

Baseline provenance. All contestants started from an identical checkout of the round-2 champion: Fable 5's default-harness engine at the tag `baseline` (commit `84711ac`), scoring 89,191 of 91,280 ASan-clean passes (97.71% S2). Of its 2,089 failing cases, 224 are `$262.agent` (multi-threaded agent) cases excluded from the target set, leaving 1,865 reachable failures over a 91,056-case target set. The budget was 6 hours plus a 30-minute gate-centric wrap-up, with the same anti-gaming machinery as the prior rounds; the secret computed-output probes passed 40/40 on both binaries for every run.

The gate. Submissions had to be set-wise no-regression against the baseline: every case the baseline passes must still pass, on both the O2 and the sanitizer binary, over the full suite, with a one-case flaky allowance. Any violation zeroes the run on the milestone board regardless of how many failures it fixed. Contestants were given the *exact* official gate checker (`check-gate.sh`) and were told, in both the every-turn driver prompt and the wrap-up prompt, to run the full gate before their final commit.

Milestones and replay scoring. The score is milestone-based: M1 = 466 failures fixed, M2 = 933, M3 = 1,399, M4 = 1,679. The primary ranking is the highest milestone reached, with crossing time as the tiebreak; the secondary ranking is final fixed count plus area under the progress curve. Crossing times come from a judge-side replay of every commit in every run — a 5,000-case sampled curve to localize each crossing, then full-suite verification at the crossing commits — with times

Table 6: Round-3 (last-mile) final results; all eight runs officially scored. "Fixed" is the official judge-side count of baseline failures fixed; "Board" is the milestone-board score after gate enforcement (a gate violation zeroes the run — five of the eight runs were zeroed). M1/M2 are replay-verified crossing times in minutes from each contestant's start. Gate regressions are counted on the O2/ASan binaries respectively.

Contestant	Gate (O2/ASan regr.)	Fixed	Board	M1 (+min)	M2 (+min)
Fable 5-ultra	PASS (0/0)	1,243	1,243	89	103
Fable 5	PASS (0/0)	1,170	1,170	124	292
Opus 4.8	PASS (0/0)	935	935	118	—[a]
GPT-5.5	regressed (4/4)	1,003	0	145	314
Gemini 3.5	regressed (100/100)	913	0	129	299
Opus 4.7	regressed (26/28)	873	0	122	—
Opus 4.8-ultra	regressed (6/8)	778	0	185	—
Opus 4.7-ultra	regressed (128/128)	1,000	0	(est.)[b]	—

[a] Opus 4.8 cleared M2 only in the final official full-suite count ($935 \geq 933$); the sampled replay curve never resolved an M2 crossing mid-run, so no crossing time is assigned. [b] Opus 4.7-ultra's replay verification was incomplete at publication; the sampled curve suggests it crossed M1 (peak scaled estimate roughly 1,077 fixed), but no verified crossing time is assigned, so its M1 is reported as estimated only. It never reached M2, and its board score is 0 under the gate regardless.

normalized to each contestant's own start. In practice a sizable fraction of the 1,865 reachable failures appears unwinnable under the staging runner: Fable 5-ultra's wrap-up accounting put the figure at roughly 620 (585 script-mode dynamic-import cases plus 35 it root-caused as unwinnable) — a contestant self-analysis rather than a judge-side measurement, but one consistent with no run exceeding 1,243 fixed. M3 was therefore out of reach in practice, and the race was decided by the earliest M2.

9.2 Results

Table 6 reports all eight runs. Wave A ran first; the lineup re-used the prior contestants on the harnesses that defined them: Fable 5 under both the default and the workflow (ultra) harness, Opus 4.8 and Opus 4.7 under the default harness, GPT-5.5 under codex at xhigh reasoning effort (verified from request headers), and Gemini 3.5 under `agy`. Wave B, the following day, added workflow variants of the two Opus models.

Fable 5-ultra won Wave A outright: gate-clean, the most failures fixed (1,243, lifting the engine from 97.71% to 99.07% S2), the fastest replay-verified M1 (89 minutes, 835 fixed at the crossing commit) and M2 (103 minutes, 969 fixed), and the highest area under the progress curve of the six replayed runs (6.1k case-hours versus 5.5k for the next best). The next-fastest M2 was its own default-harness sibling, Fable 5, at 292 minutes — 3 hours 9 minutes later; Opus 4.8 and Opus 4.7 never crossed M2 on the replay curve at all. Final engine pass rates were tightly bunched (98.60–99.07% S2 across Wave A), but the board was not: three runs (GPT-5.5, Gemini 3.5, Opus 4.7) shipped regressions and scored zero despite fixing 873–1,003 cases each. Code footprints varied widely — from Opus 4.8's +1,432/−516 lines over 56 commits (the smallest) to Fable 5-ultra's +13,778/−933 over 89 commits and Gemini 3.5's +100,122 inserted lines (mostly generated Unicode casing tables). Among the Claude runs, output/novel/billable tokens were 3.57M/16.46M/20.5M for Fable 5-ultra, 2.25M/6.17M/8.7M for Fable 5, and 2.43M/4.87M/7.4M for Opus 4.8.

Wave B made the gate's verdict the story: both workflow variants of the Opus workflow variants

regressed and scored zero. Opus 4.8-ultra finished at 98.55% S2 (89,961 passes), fixing 778 cases over 47 commits (+1,237/−261 lines), but shipped 6 regressions on the O2 binary and 8 under ASan; it crossed M1 at +185 minutes (verified, 570 fixed at the crossing commit) and never reached M2. Opus 4.7-ultra finished higher — 98.67% S2, 1,000 fixed over 50 commits (+4,367/−1,155) — and regressed harder: 128 cases on each binary. Its own wrap-up log declared "Gate: PASS … 98.93% (2968/3000)" — a verdict from a 3,000-case *sample* gate that missed every one of the 128 regressions the full official gate found. The final board therefore ranks three runs: Fable 5-ultra first (1,243 fixed, M2 at +1h43m), Fable 5 second (1,170, M2 at +4h52m), and Opus 4.8 third (935, M2 only at the final count). Five of the eight runs were zeroed by the gate.

9.3 The harness flip: parallel workflow wins the many-small-fixes regime — for the model that can hold the gate

The strongest cross-contest finding of the study is that harness fit is task-shape-dependent. In the greenfield regime — the round-2 Fable 5 addendum — the workflow (ultra) harness *cost* Fable 5 0.92 points against its own default run (96.79% versus 97.71%, Section 5). In round 3, the same harness on the same model was dominant: Fable 5-ultra reached M2 at 1 hour 43 minutes, versus roughly five hours for every other run that crossed it. Greenfield engine construction rewards a coherent single-threaded architecture; the last mile is a large pile of small, independent spec fixes, which is exactly what a parallel worktree fan-out is shaped for.

Wave B scopes this finding sharply. The workflow harness was dominant *for Fable 5*; it did not transfer to the Opus models. Opus 4.8-ultra and Opus 4.7-ultra ran the same parallel-workflow harness on the same task and both regressed out of the board — and neither converted the fan-out into Fable 5-ultra's speed (one verified M1 at +185 minutes; no verified M2 between them). The only run that converted workflow speed into a gate-clean win was Fable 5's. "Workflow dominant in this regime" must therefore be scoped to model capability: orchestration amplifies whatever discipline the model already has, and a harness that multiplies a careful model's throughput multiplies a careless model's regressions.

The replay shows the mechanism directly. Fable 5-ultra prepared merge hygiene 14 minutes into the run (a merge-driver commit keeping its own build artifacts during area-branch merges), then fanned out across seven area branches in round one of its internal plan (`wt/{temporal,buffers,jsonstr,asyncmod,misc,class,langtail}`) and six more (`wt2/…`) in two later rounds, for 21 merge commits above the baseline. Its M1→M2 transition was a single 2.5-minute merge train: in minutes 89–92 of its run it merged `wt/langtail`, a cross-branch semantic-conflict fix, then `wt/buffers` and `wt/temporal`, jumping the judge's sampled fixed-estimate from roughly 237 to roughly 913; the verified M1 crossing is the conflict-fix commit itself, and M2 followed 13.6 minutes (817 seconds) after M1, at a later parser-fix commit rather than in the train itself. It also tracked its own progress against a self-defined pool of 1,280 winnable cases (the 1,865 reachable minus 585 it identified as unwinnable script-mode dynamic-import cases), banking pool counts in rebuild commits (1,015 → 1,069 → 1,091 → 1,121 → 1,245, each asserting "0 regressions"). And it took the gate seriously: sampled gate checks at three checkpoints, plus the full one-hour gate launched *detached* at roughly +3h31m so it would survive the iteration-window kill — a run that surfaced two real regressions (a too-broad legacy `fn.caller` getter and a TypedArray integer-index descriptor change), both fixed in dedicated "GATE FIX" commits before the final submission.

9.4 The gate split the field exactly on instruction-following

The no-regression gate turned out to be a clean instruction-following discriminator. The three gate-clean runs are precisely the three that either completed the full gate before shipping (the winner, detached) or kept the tested tree identical to the shipped tree; the five zeroed runs (GPT-5.5, Gemini 3.5, Opus 4.7, Opus 4.8-ultra, Opus 4.7-ultra) share one dominant failure pattern: *gating on a sample instead of the full suite*. In Wave A, every recorded `check-gate.sh` invocation across the three zeroed runs used the `--sample` mode (at most 3,000 of 91,280 cases, about 3.3%), despite the brief's explicit instruction to run the full gate before the final commit and despite every sampled run printing "(PARTIAL — sample mode; official gate is the full suite)."

The Wave A per-run autopsies differ in flavor. GPT-5.5 never invoked the gate checker at all during its six-hour build phase — the only "check-gate" strings in its session logs are the brief text itself — substituting directory-scoped diffs against baseline failures only in the areas it touched (JSON, ArrayBuffer, DataView, String). Its first-ever gate execution came in the wrap-up at 6 hours 18 minutes into a 6.5-hour run; the sample immediately reported two regressions, which it fixed and re-verified (sampled) three times, but the four official regressions per binary lay outside the 3.3% sample, and by gating only at the end it left no window in which the prescribed one-hour full gate could ever fit.

Gemini 3.5 was the most diligent *sampled* gater of the three — dozens of invocations at sample sizes 100–3,000, including a mid-run sampled catch-and-fix — and its wrap-up did rebuild and recommit both binaries after its final three source commits (a dynamic-import module rewrite, parser changes, and an `Array.prototype.toString` change, landed in its final logged iteration, minutes before its wrap-up), then ran one more 3,000-case sampled gate that reported PASS. But that 3.3% sample is the only check those late changes ever faced: the 100 regressions per binary they introduced lay outside it, and the full gate that would have caught them was never run.

Opus 4.7's record is thinner still: its wrap-up log self-reports sampled-gate PASS claims, but the surviving local logs contain no recorded `check-gate.sh` invocation at all (its main-run log was captured only as a stub), and it shipped 26/28 regressions.

A build-level autopsy of the Opus 4.8 pair — the same model under the two harnesses, with opposite gate outcomes — sharpens the mechanism (full write-up in `docs/AUTOPSY-opus48-ultra.md`). The workflow variant's 14 regressed case-runs collapse to four test files injected by three ordinary commits landed in a nine-minute consolidation burst, in which the orchestrator hand-re-ported subagent worktree code onto `main`: one dropped guard is directly visible in the bundle (an immutable-buffer `TypeError` check present on the `fix/typedarray` branch but absent from the consolidated commit), and one unclamped month index reads past a 12-entry table — silent under O2 but fatal under the sanitizer, which alone explains the run's 6-versus-8 split. Sixteen subsequent sampled gates all passed because `check-gate.sh --sample` draws a *deterministic* evenly-spaced subset that never contained the damaged cases; a mid-run sampled catch-and-fix of two *other* burst regressions manufactured false confidence.

Notably, neither Opus 4.8 run ever completed a full gate — the default run attempted one and the driver's 45-minute turn timeout killed it. Across the whole field only the eventual winner, Fable 5-ultra, completed the full 182,560-case gate, by launching it *detached* at roughly +3h31m so it survived the turn kill — the exact escape hatch the turn budget otherwise forbade.

The Opus 4.8 default's clean ship was instead earned by loop hygiene rather than heavier checking: its strictly linear edit–gate–commit cycle kept the tested tree identical to the shipped tree, and when it did regress once mid-run, its very next sampled gate caught the damage in the just-changed delta and a named fix landed fifteen minutes later. The harness lesson generalizes: parallel orchestration broke the identity between verified and shipped code, and a deterministic

sample cannot police that gap no matter how often it is re-run.

Wave B supplied the failure mode's sharpest instance. Opus 4.7-ultra did not skip gating — its wrap-up log declared "Gate: PASS … 98.93% (2968/3000)" — but that verdict came from a 3,000-case sample that missed every one of the 128 regressions the full official gate found on each binary. It gated, got a green light, and shipped 128 regressions anyway: a sampled gate that says PASS is not a gate. Opus 4.8-ultra's 6/8 regressions follow the same pattern at smaller scale. The discipline split is also generational: the only three gate-clean runs in the contest are the two Fable 5 runs and Opus 4.8. The instruction was identical, repeated every turn, and mechanically checkable; whether a run obeyed it fully predicted whether the run scored.

9.5 Self-measurement reliability

The greenfield rounds' contestants self-estimated close to their official scores; the largest overstatement, by Fable 5 in the round-2 addendum, was about 0.7 percentage points (Section 5). Round 3 produced the study's first wild miss: Gemini 3.5 self-claimed 1,436 failures fixed, against an official 913 — a 57% overstatement — while simultaneously shipping 100 regressions on each binary. The claim was not merely stale (its committed binaries lagged its source); its own sampled measurements diverged from the judge-side replay throughout the back half of the run. The Claude-family pool counts embedded in Fable 5-ultra's commit subjects, by contrast, tracked the official replay closely, and its final self-asserted "0 regressions" matched the official gate verdict. Where self-measurement is load-bearing — as it is for any agent deciding whether it is safe to ship — this reliability gap is itself a capability difference, not just a bookkeeping one.

9.6 Operational incidents

Four incidents are recorded for round 3, handled under the standing fairness rule (Appendix B, incidents 10–13). (1) Both Wave B runs were stalled mid-run by a shared subscription session limit (an approximately 88-minute shared stall); makeup windows were granted (162 minutes for Opus 4.8-ultra, 88 minutes for Opus 4.7-ultra). (2) Gemini 3.5 was quota-throttled by HTTP 429 storms (on an Ultra plan with roughly 5-minute reset cycles) from about three hours in; its authentication held, a watchdog false-flagged it once (cleared), and it finished its full window. (3) A judge-side scoring session wedged in its terminal multiplexer and was relaunched, with no effect on results. (4) The replay pipeline needed two fixes — v1 scored under the sandbox wrapper while the contest ran bare, and v2 hit checkout conflicts on tracked binaries, resolved with a force-checkout — after which all replays were rerun; all crossing times reported above are from the rerun.

10 Round 4: Nitro — racing the champion engine for raw speed under a zero-regression gate

Round 3 measured finishing; round 4 measures a different axis entirely: making a finished engine *fast* without breaking anything it already does. Every contestant started from the same checkout of the round-3 champion engine — Fable 5-ultra's last-mile tree at 99.07% of test262 — and had four hours plus a 30-minute wrap-up to maximize its throughput. The score is the geometric-mean speedup over the baseline across seven workload classes, measured on a *hidden* workload suite the contestant never saw, and gated by the same full-suite zero-regression contract as round 3: any correctness regression zeroes the run. In contest numbering this is round 4; the field, harness assignments, and per-model CLIs are identical to round 3 (Section 9). Wave A (six runs) launched first and Wave B (the two Opus workflow variants) followed five and a half hours later, under the

same shared-subscription constraints as before. Six of the eight runs shipped gate-clean — including both non-Claude contestants that the round-3 gate had zeroed — and the only two zeroed runs were the two Claude Opus 4.7 variants, one of them the fastest raw engine in the field.

10.1 Design

Baseline provenance. All contestants started from an identical checkout of the round-3 champion at the tag `baseline` (commit `46935fb`): the engine Fable 5-ultra's last-mile run produced, passing 99.07% of the 91,280-case suite. The budget was 4 hours of wall-clock time plus a 30-minute wrap-up; the scored tree is, as in every round, the final committed `main` at collection.

Workloads and the hidden suite. Speed is measured on seven procedurally generated workload classes: `numeric` (arithmetic kernels), `strings`, `propchurn` (property creation, deletion, and shape churn), `calls`, `regex`, `gcpressure`, and `parse`. A public development suite, committed in-repo (`bench/nitro-public/`), was given to contestants to tune against. The *official* suite was generated on the timing box, after collection, from a seed withheld until then; contestants never saw it. The official public leg was timed on the literal committed development files.[1] The ratio of a run's public-suite geomean to its hidden-suite geomean is reported as an overfit metric: a ratio well above 1 would indicate tuning to the published instances rather than to the workload class. Each workload prints a computed result checksum, and a workload counts as valid only if the checksum is correct on *every* repetition — an engine cannot buy speed with wrong output. A wrong checksum or timeout on any rep marks that workload `FAIL` (excluded from the geomean) and the run's bench verdict `INVALID`, which voids the run's speed score on the board; in the event this was moot, as all eight runs were checksum-valid on 7/7 workloads on both suites.

Timing methodology. All official timing ran on a single dedicated-core Hetzner ccx23 box. For each workload, baseline and candidate binaries were run interleaved (B, C, B, C, …) for 5 repetitions each, to cancel thermal and cache drift; the per-engine median wall time defines the speedup, with a 300-second per-rep timeout. The aggregate score is the geometric mean of the per-workload speedups over valid workloads, so no single class dominates the headline number. All eight runs delivered 7/7 valid workloads with every checksum correct on every rep.

The gate, version 2. Submissions had to be set-wise no-regression against the baseline's full-suite results — every case the baseline passes must still pass, on both the O2 and the sanitizer binary, over all 91,280 cases — with a flaky carve-out that, in the event, was empty (no case was excluded). Any violation zeroes the run's board score regardless of raw speed. The contestant-side gate kit was upgraded in direct response to the round-3 autopsies (Section 9): sampled gate checks now draw *randomized* subsets rather than the deterministic evenly-spaced subset whose fixed blind spots round 3's zeroed runs re-tested over and over (the script's header comment still carried the stale round-3 description; the runtime banner printed the correct randomized-sample one), and the kit included a launcher that runs the full two-binary gate *detached* from the agent's turn — the exact escape hatch that only round 3's winner had discovered for itself. The board score is the hidden-suite geomean if the gate passes, and 0 otherwise.

[1] The generator, as run, derived its per-class random streams through Python's per-process salted `hash()`, so a suite is *not* byte-reproducible from its seed alone; the scored suites are pinned by archived copies of the exact workload files (`results/official/nitro-timing-box/`) rather than by the seeds. A first public-leg timing pass accidentally used a fresh seed-1000 regeneration instead of the committed files; it was caught in audit and the public leg was re-timed on the committed files with the same binaries, box, and methodology. The two legs agree within noise and the hidden-suite board was never affected. Two of the seven classes (`propchurn`, `regex`) have seed-invariant reference checksums; their per-seed freshness is in constants and timing shape only.

Table 7: Round-4 (nitro) final results. "Hidden" is the official geometric-mean speedup over the baseline engine on the hidden workload suite; "Public" the same on the published development suite; "Overfit" is public/hidden. "Board" is the score after gate enforcement (a gate violation zeroes the run). Gate regressions are counted over the full 91,280-case suite on the O2/ASan binaries respectively. All eight runs were checksum-valid on 7/7 workloads.

Contestant	Harness	Gate (O2/ASan regr.)	Hidden	Public	Overfit	Board
Opus 4.8-ultra	workflow	PASS (0/0)	4.79x	4.88x	1.018	4.79
Opus 4.8	default	PASS (0/0)	4.20x	4.20x	1.000	4.20
Fable 5	default	PASS (0/0)	3.94x	3.87x	0.982	3.94
Fable 5-ultra	workflow	PASS (0/0)	3.80x	3.76x	0.990	3.80
Gemini 3.5	agy	PASS (0/0)	2.66x	2.65x	0.997	2.66
GPT-5.5	codex	PASS (0/0)	1.38x	1.41x	1.015	1.38
Opus 4.7	default	regressed (9/9)	5.11x	5.06x	0.990	0
Opus 4.7-ultra	workflow	regressed (7/5)	4.33x	4.23x	0.976	0

Table 8: Per-workload hidden-suite speedups (median-of-5, interleaved, checksum-validated). The two gate-zeroed runs are below the rule.

Contestant	numeric	strings	propchurn	calls	regex	gcpressure	parse
Opus 4.8-ultra	8.96	4.55	8.71	5.82	3.15	4.38	2.02
Opus 4.8	7.85	4.12	7.01	4.98	2.28	4.27	2.09
Fable 5	7.79	2.82	7.43	4.06	2.62	4.15	2.06
Fable 5-ultra	7.53	2.89	7.03	4.38	2.10	4.13	1.96
Gemini 3.5	3.12	2.18	4.83	3.21	2.14	3.02	1.37
GPT-5.5	1.32	1.54	1.32	1.79	1.47	1.41	0.99
Opus 4.7	9.29	4.88	9.00	5.64	4.47	4.49	1.98
Opus 4.7-ultra	6.60	4.33	6.80	5.25	3.54	3.84	2.07

10.2 Results

Table 7 reports the board and Table 8 the per-workload hidden-suite speedups. The lineup matched round 3: Fable 5 under both the default and workflow (ultra) harnesses, Opus 4.8 and Opus 4.7 under both as well (the workflow variants in Wave B), GPT-5.5 under codex (v0.135.0, xhigh reasoning effort), and Gemini 3.5 under `agy`.

Opus 4.8-ultra won at 4.79x, gate-clean, leading the gate-passing field on six of the seven workloads and trailing only on `parse` (2.02x versus its default sibling's 2.09x). Opus 4.8 took silver at 4.20x and Fable 5 bronze at 3.94x, with Fable 5-ultra close behind at 3.80x. Gemini 3.5 (2.66x, in only three long driver iterations) and GPT-5.5 (1.38x, with `parse` essentially unimproved at 0.99x) were slower but clean: both non-Claude contestants that the round-3 gate had zeroed shipped 0/0 this time. The fastest raw engine in the field did not win: Opus 4.7 reached 5.11x — ahead of the gold medalist by 0.32x — and was zeroed with 9 regressions on each binary; Opus 4.7-ultra reached 4.33x and was zeroed with 7 (O2) and 5 (ASan). The hidden suite did its job quietly: overfit ratios spanned 0.976–1.018, so no run's public-suite tuning bought it more than about 2% of phantom speedup, and the public and hidden boards would have ranked the field identically.

The runs converged on a recognizably common optimization menu — property/shape caches

and inline caches, arena allocation, regex prefilters and first-character bitmaps, ASCII fast paths in the lexer — with the spread coming from how much of the menu each run landed and gated. The largest per-workload speedups came in `numeric` and `propchurn` (up to 9.29x and 9.00x, both by the zeroed Opus 4.7); `parse` was the stubbornest class, with no run exceeding 2.09x.

10.3 Velocity without verification: the Opus 4.7 family is 0-for-4 on gates

Across the study's two gated contests, the Claude Opus 4.7 family has now been zeroed four times in four runs — 26/28 and 128/128 regressions (O2/ASan) for its default and workflow runs in round 3, and 9/9 and 7/5 here — while posting front-of-field raw production each time. (The blind replication of this round later extended the record to 0-for-6; Section 11.5.) The pattern is sharpest in this round because the metric makes it quantitative: Opus 4.7 built the fastest engine of the field, 5.11x against the winner's 4.79x, and converted it into a board score of zero.

The per-run records show the same verification gap as round 3. Opus 4.7's surviving gate artifact is an 8,000-case *sample* PASS in its own `fullgate.log`; the full official gate found nine regressed case-runs per binary, spanning frozen- and non-writable-array `push` length semantics, a sloppy-`arguments` `concat` case, a modulus edge case (`S11.5.3_A4_T2.js`), and two sloppy-mode function-code cases. Its last act was a commit titled "GATE FIX" (`d649a4d`, roughly 50 minutes past its window end and 20 minutes into the wrap session) repairing a case-insensitive regex bug that its own first-character-bitmap optimization had introduced. Per the standing rule the scored tree is the final committed `main` at collection, so the late fix was accepted and gated — and the tree still regressed 9/9. Opus 4.7-ultra's five regressions shared by both binaries are plain semantic breaks (compound-assignment `mod-whitespace.js`, modulus `S11.5.3_A4_T6.js`, and a `for-in head-lhs-let.js` case, each in both modes where applicable); a `RGI_Emoji.js` pair regressed on the O2 binary only.

What makes the family pattern notable in this round is the rest of the field. Round 3's gate split fell partly along tooling lines — the deterministic sampled gate and the turn budget made the full check structurally hard to run. Round 4 removed those excuses: the kit's samples were randomized and the full gate came with a detached launcher, and six of eight runs — including GPT-5.5 and Gemini 3.5, both zeroed in round 3 — shipped 0/0. The improvement should not be over-read as a verification story, however. The collected logs support a precise census: three of the eight runs completed a clean full-suite self-gate before the verdict (Fable 5 inline-by-choice, Fable 5-ultra, and Opus 4.8 detached, its verdict landing after the wrap), and all three shipped clean; the other five shipped on sampled evidence alone — and they split three clean (including the *winner*, Opus 4.8-ultra, whose wrap ran an 8,000-case sample and recorded it as "partial evidence, not the full-suite proof") against two zeroed (both 4.7s). GPT-5.5's own detached full gate did complete — with a spuriously `REGRESSED` verdict listing an implausible 17,683 all-ASan failures that the official 0/0 gate contradicts — and it shipped anyway, extending round 3's self-measurement-reliability thread. So the round-4 gate discriminated on what it actually measures — whether the shipped tree regressed — not on whether the shipper had proven it would not. What distinguishes the 4.7 family is that its sampled evidence was wrong both times: on both harnesses, the final sample-PASS artifact preceded a tree that the full gate then failed.

10.4 How Fable 5 lost: clock management, not capability

Fable 5 entered as the engine's author lineage and the study's conformance champion, and finished third at 3.94x. A commit- and log-level autopsy (`docs/AUTOPSY-nitro-fable.md`) finds the loss was driven by time allocation under gate risk-aversion, not by optimization skill.

The run's shape: an opening burst of performance commits over 41 minutes (minutes 15–56 of the window), then roughly 71 minutes (minutes 58–130) spent waiting idle for the full 182,560-case gate verdict with the window clock running — the gate itself ran through the detached launcher, but the run chose to hold for it rather than optimize alongside it — then a deliberate freeze: "post-gate freeze" status commits at minutes 133 and 142, with run-log iterations 26 through 103 (78 turns) spent re-affirming the freeze rather than optimizing. Its log states the reasoning verbatim: "the full gate takes ~75 min… any engine change now would ship un-gated — exactly how five of eight contestants zeroed out last time." The caution was evidence-based — one of its own bugs had survived five consecutive sampled gates before the full gate caught it, and the log prices the trade explicitly ("the gated ~3.5x is worth more than an un-gated ~3.6x") — and directionally vindicated, since Opus 4.7 did precisely what Fable 5 refused and was zeroed. But it was mispriced: Opus 4.8 optimized continuously until 11 minutes past the window end behind cheap sampled gates and straddled the deadline with the full gate, and took silver gate-clean. The freeze was compounded by a deadline mis-anchor: the run believed its window had started 56 minutes earlier than it had (matching box-preparation file timestamps), so it scheduled its gate and freeze against a phantom deadline 56 minutes early, surrendering roughly a quarter of its real budget. Net, Fable 5 spent on the order of an hour of its four-hour window changing engine code, with its last in-window performance commit at minute 56 — more than three hours before the real deadline — while the Opus 4.8 and Opus 4.7 default runs committed performance work up to and past the window end.

Capability was demonstrably not the constraint. A still-live optimizer process from its final iteration (launched at minute 232) resumed work after the window and, in roughly twenty minutes beginning ten minutes after the window end, committed four performance commits that lifted the self-benched geomean from 3.33x to 4.26x. Under the standing final-committed-`main`-at-collection rule — the same ruling that accepted Opus 4.7's late commits — those four commits are part of Fable 5's scored tree: the official 3.94x was timed on the engine including them, and the official 0/0 gate covered them. The post-window process thus *recovered* part of the podium margin the freeze had surrendered; the tree Fable 5 had deliberately frozen in-window self-benched only 3.33x. The process then went one step further, leaving *uncommitted* edits in the working tree mid-wrap — a member-property inline cache, the exact architecture Opus 4.7 rode to 5.11x, plus further regex work — and here the wrap-up did the disciplined thing: it stashed those edits as foreign and ungated, proved the clean-HEAD rebuild bit-identical to the gated binary, re-gated, and shipped without them (the 4.26x final bench ran on the clean HEAD, excluding the inline cache). Fable 5 generated the winning idea and excluded it — but only that uncommitted batch — from its own scored tree.

The per-workload decomposition localizes the cost. Against Opus 4.8, Fable 5 actually *won* `regex` (2.62x vs. 2.28x) and `propchurn` (7.43x vs. 7.01x); in log space, `strings` accounts for 87% of the deficit and `calls` for 47%, offset by those wins. Both are exactly the leads its own log names as abandoned at the freeze ("regex inner loop, string churn, per-call env setup"). The counterfactual is tight: substituting only Opus 4.8's `strings` and `calls` numbers into Fable 5's per-workload vector yields a geomean of 4.29x — ahead of Opus 4.8's actual 4.20x. One or two afternoon-sized optimizations separated Fable 5 from the gate-clean silver or better.

The general lesson is about risk posture, not skill. In a conformance race the loss function is monotone: every verified fix is banked, and verification time is never wasted — the policy Fable 5 executes instinctively, and which won it round 2. In a speed race under a zero-regression gate, verification is insurance whose premium is paid in the only currency the score counts, and the optimal policy decouples proof from progress: optimize to the wire behind cheap randomized samples, and schedule the expensive full proof so that it does not consume optimization time. Fable 5 imported the conformance policy wholesale into a task whose payoff matrix had inverted. Task

shape selects for risk posture as much as for capability — and a model's dominant trait is a strategy prior it will re-apply even when the payoff matrix no longer rewards it.

10.5 The workflow harness's second straight win — earned at the bench, not at the gate

Opus 4.8-ultra's gold is the workflow (ultra) harness's second consecutive contest win, after Fable 5-ultra's in round 3 — and this time on a model whose round-3 workflow variant had regressed out of the board. But the win was not earned by the endgame protocol the round-3 analysis (Section 9) identified as correct for a hard no-regression contract, and the record is worth stating plainly. Opus 4.8-ultra optimized to the wire — and ten minutes past it: its final performance commit, `e51a204`, enabled `-flto` on the O2 target 10.5 minutes after its window end, accepted under the standing collection rule and directly relevant to the officially timed binary — behind eight in-window randomized sampled gates of 2,000–3,000 cases each. It never launched the detached full gate. Its own wrap-up log says so verbatim ("`~/fullgate.log` didn't exist — I'd never launched it"), then ran a final 8,000-case sampled gate, recorded the PASS in its head commit `09253f3`, and noted for the record that this was "a *sampled* gate (8000 cases) — partial evidence, not the full-suite proof." The only full-suite 0/0 proof for the winner is the judge-side gate. The contrast with the zeroed Opus 4.7 is therefore narrower than a verification-discipline story: both shipped on the same class of evidence — a final sampled-gate PASS — and the gate's opposite verdicts measured their trees, not their proofs. The protocol itself was executed, correctly, by others: Opus 4.8 (default) optimized continuously behind sampled gates and launched the full gate detached in its wrap-up, after its last performance commit — a deadline-straddling gamble whose verdict (PASS) landed only after its wrap, vindicated by the official 0/0 — and both Fable 5 runs completed full self-gates before shipping.

The round-4 evidence sharpens the study's harness finding rather than reversing it. Round 2 showed the workflow harness is not a free improvement (it cost Fable 5 0.92 conformance points); round 3 showed it amplifies whatever discipline the model brings (a gate-clean win for Fable 5, regressions for both Opus workflow variants); round 4 adds the complement: with the verification tooling fixed — randomized samples that cannot be fooled by re-testing a fixed subset, and a full gate that can run outside the agent's turn — the same model-plus-harness pairing that regressed out of round 3 won round 4 outright. Orchestration multiplies throughput in every regime; what round 4 does *not* demonstrate is that the winner's verification kept up with that throughput — its clean gate is a measured fact about the tree it shipped, established judge-side, not a proof it performed. Given the identical tooling, the Opus 4.7 runs were zeroed on both harnesses. The harness amplifies; the gate still discriminates on the model — on what it ships, whether or not it has proven it. Round 5's blind replication supplies the postscript: the identical configuration repeated the identical sampled-evidence endgame and was zeroed by two cases out of 91,280 (Section 11.4).

10.6 Operational incidents and threats to validity

Ten incidents are recorded for round 4 (Appendix B, incidents 14–23).

1. The first launch of the judge-side gate queue was invalid: a relative `--exec-wrapper` path caused all 91,280 cases to error at the runner on every leg, so every run showed the same impossible count of roughly 90,436 "regressions." The absurd identical numbers tripped a sanity check before any verdict was recorded; the path was absolutized and the queue fully relaunched. This is the second occurrence of this bug class in the series' judge tooling (the

round-3 flaky-list queue hit the same trap), and it was judge-side only — no contestant artifact or verdict was affected.

2. The freshly provisioned timing box's package index was unpopulated and engine builds failed until it was updated; this preceded all official timing.

3. Opus 4.7's post-window "GATE FIX" commit was accepted under the standing collection rule, as described above — the verdict stands on the tree the contestant shipped (whether the window-end tree alone would also have regressed was not separately tested).

4. Opus 4.7-ultra hit a Claude session limit at iteration 6; iterations 7–10 bounced and iteration 11 started with roughly 11 minutes of window left.

5. Both Wave B wrap-up sessions fired roughly 10.8 hours after their windows ended (a scheduling fault); Opus 4.8-ultra's late wrap ran a final 8,000-case *sampled* gate and recorded that PASS (its detached full gate had never been launched), while Opus 4.7-ultra's produced no captured log — though its tree gained two engine fixes and a rebuild minutes before collection, see incident (9).

6. The stray process on Fable 5's box (previous subsection) committed four post-window performance commits that remain in the scored tree under the collection rule; the contestant's own wrap-up handled the process's later *uncommitted* edits — stash, provenance re-check, clean-HEAD re-gate — and only those are excluded.

7. The Gemini 3.5 and Opus 4.7-ultra trees were dirty at collection; the collector committed "final state" snapshots, and contestant-authored commit counts (10 and 39 respectively) exclude that snapshot.

8. Opus 4.7-ultra's gate legs were the last in the judge queue to finish (ASan leg completed roughly 18 hours after the Wave B window ended, final counts 7 O2 / 5 ASan; its ASan regression set is a strict subset of its O2 set).

9. Because collection occurred roughly 17 hours (Wave A) and 11.7 hours (Wave B) after the windows ended, the final-`main`-at-collection rule admitted post-window commits broadly, and they were disclosed and ruled on uniformly: post-window *engine* commits exist in the scored trees of five of the eight runs — Opus 4.7 (seven, including the GATE FIX), Fable 5 (the four stray-process commits), Opus 4.8 (one lexer fast path, +11 minutes), Opus 4.8-ultra (the `-flto` commit, +10.5 minutes), and Opus 4.7-ultra (five in the in-flight final turn plus the two fixes and rebuild before collection) — while Fable 5-ultra, Gemini 3.5, and GPT-5.5 committed none. All were gated; every verdict stands on the shipped tree.

10. The first official public-suite timing leg accidentally benched a fresh seed-1000 regeneration rather than the committed development files (the generator's per-class streams pass through Python's salted `hash()`, so regeneration is not byte-stable; five of seven regenerated workloads differed by checksum). The public leg was re-timed on the literal committed files with the same binaries, box, and methodology; the two legs agree within noise, the field ranking is identical on all three suites, and the hidden-suite board was never affected. Table 7's Public and Overfit columns use the committed-suite leg, and all suites, drivers, and raw logs are archived under `results/official/`.

Threats to validity are mostly inherited from the earlier rounds (single run per configuration; operator-in-the-loop incident handling), with two specific to timing. First, all official timing ran on one dedicated-core box; the interleaved B/C ordering, median-of-5 aggregation, and per-rep checksums control drift and wrong-output speed within the box, but cross-box variation is unmeasured. The podium separations are large (14% and 7% between adjacent medalists), but the ordering between Fable 5 and Fable 5-ultra (3.94 vs. 3.80, a 4% gap) should be treated as provisional given a single run and a single timing box. Second, the hidden suite bounds overfitting to the published *instances* (ratios 0.976–1.018), but both suites come from the same generator, so specialization to the generator's workload *shapes* — shared by every contestant equally — is not bounded by this metric. Third, Wave A and Wave B ran on different days under different in-window conditions: Wave A ran four concurrent Claude sessions on the shared subscription against Wave B's two, and the Wave B wrap-ups fired 10.8 hours late, so a Wave B wrap could in principle have acted on a completed detached full-gate verdict that no Wave A wrap (fired roughly 30 minutes after window end, before any full gate could finish) could see — in the event, neither Wave B wrap had such a verdict available. All engines were timed on the same box under the same methodology, and the gate is day-independent.

11 Round 5: Nitro, replayed — a blind replication with a single-variable prompt change

Round 4 left two questions that only a rerun could answer. The first was a treatment question: would a revised system prompt change the time-allocation pathology that the Fable 5 autopsy (Section 10) had documented — the early freeze, the inline gate hold, the refusal to decouple proof from progress? The second was a methodological one the whole series had been accumulating: with one run per configuration, how much of any round's board is the model and how much is the draw? Round 5 answers both by replaying round 4 *blind*: the same field, the same baseline engine, the same brief, budgets, wave structure, and on-box labels, on freshly re-imaged boxes that carried no trace that a first running of the race had ever happened. One variable was changed on purpose — the two Fable 5 boxes ran a revised system prompt (an alternate variant of the harness's production system prompt) — and everything else was held as close to constant as live infrastructure allows. Wave A (six runs) launched first, and Wave B (the two Opus workflow variants) roughly four and a half hours after Wave A, with a 10-minute makeup window granted to both Wave B runs for a harness-side outage (below).

The headlines: Fable 5-ultra, on the revised prompt, won at 5.19x — the highest gated speedup of the entire series — with a two-full-gate endgame. The round-4 champion configuration repeated its round-4 strategy verbatim and was zeroed by a two-case regression. The Opus 4.7 family fell to 0-for-6 on gates. And the six control runs swung, on identical workloads, by amounts as large as — and in one case roughly twice — the treatment effect, which is itself a finding, and the one that bounds all the others.

11.1 Design

Replication fidelity. Every contestant again started from the byte-identical round-3 champion checkout at tag `baseline` (commit `46935fb`, 99.07% of test262), with the same brief, the same 4-hour window plus 30-minute wrap-up, the same per-model CLIs and harness assignments, and the same on-box run labels as round 4. The contestant boxes were the same machines, re-imaged and wiped (one box, GPT-5.5's, had been deleted after round 4 and was rebuilt, its repository sourced

from a sibling contestant box's pristine baseline preparation); a dedicated pre-launch blindness audit initially *found* residual round-4 tells, which were fixed, and launch was held until the audit returned clean. No transcript, commit, or log from any of the eight runs shows awareness that the race had been run before. The brief was byte-identical, including its cautionary line about round 3's sampled-gate disqualifications, which appears in both rounds' logs — it describes round 3, not a leak of round 4 — and including its instruction to launch the detached full gate "no later than 60 minutes before your deadline." The judge and timing boxes were reused from round 4 under a strict separate namespace, leaving all round-4 records untouched; the *hidden* workload suite was regenerated fresh from a new withheld seed, and the official public leg was timed on the literal committed development files from the start (the procedure round 4's audit had arrived at). The gate is unchanged: set-wise no-regression against the baseline's full-suite results over all 91,280 cases on both the O2 and ASan binaries, flaky carve-out empty in the event, any violation zeroing the run.

The single intended variable. The two Fable 5 boxes (default and workflow harness) ran a revised system prompt — an alternate variant of the harness's production system prompt — whose presence was verified in the live process environment on both Fable 5 boxes, and whose absence was verified on all six control boxes, before launch. The variant's identity is recorded in the study's private provenance log and is deliberately not described further here. The six other runs are controls in the strict sense: nothing about them was intentionally changed from round 4 except the shared CLI-version pin stated below (which the variant required fleet-wide).

Confounds, stated up front. Three things did change besides the prompt, and every cross-round comparison in this section must be read through them. First, all six Claude boxes ran Claude Code 2.1.165 where round 4 ran 2.1.161 — uniform *within* round 5, so it does not confound the in-round board, but it sits in every Claude-versus-itself comparison across rounds, including both halves of the Fable 5 A/B. Second, round 4's accidental phantom-deadline stimulus — box-preparation file timestamps 56 minutes older than the actual launch, which round 4's Fable 5 used to mis-anchor its deadline — was structurally weaker this round (preparation finished roughly 20–25 minutes before launch), and both Fable 5 runs held *correct* deadline beliefs; any improvement in deadline behavior is therefore confounded by the weaker trap, not attributable to the prompt alone. Third, and dominating both: $n = 1$ per cell. The design can detect behavioral regime changes, which are documented from logs and commits below; it cannot support causal claims about geomean deltas, and Section 11.6 quantifies why.

11.2 Results

Table 9 reports the round-5 board with the round-4 results alongside, and Table 10 the per-workload hidden-suite speedups. All sixteen timing legs (eight runs, hidden and public suites) were checksum-valid on 7/7 workloads on every repetition.

Fable 5-ultra won at 5.19x, gate-clean — the highest gated speedup of the series, ahead of round 4's winning 4.79x — on the strength of `calls` at 9.22x (no run in either round had exceeded 5.82x on that class), `numeric` at 9.70x, and `propchurn` at 8.80x. Opus 4.8 took silver at 3.41x, Fable 5 bronze at 3.15x, and GPT-5.5 fourth at 1.27x with nearly the same conservative profile as its round-4 run. Four of the eight runs were zeroed — double round 4's count — including the round-4 champion configuration, Opus 4.8-ultra, whose 4.56x raw (the field's second-fastest) was forfeited by exactly two regressed case-runs out of 91,280. The hidden suite again found essentially no public-suite overfitting: ratios spanned 0.973–1.017 (round 4: 0.976–1.018), and the rank ordering of all eight contestants is identical on the hidden and public suites in both rounds. `parse` remained the stubbornest class — no run in either round has exceeded 2.09x on it.

Table 9: Round-5 (nitro replay) final results, with round-4 results alongside. "Hidden"/"Public" are the official geometric-mean speedups over the baseline engine on the hidden and committed public workload suites; "Board" is the score after gate enforcement (a violation zeroes the run). Gate regressions are counted over the full 91,280-case suite on the O2/ASan binaries respectively. The round-4 column gives that round's hidden geomean and board score (Table 7). †Ran the revised system prompt (an alternate variant of the harness's production system prompt); all other runs are unchanged controls. [a]Both Wave B runs (Opus 4.8-ultra, Opus 4.7-ultra) hit a harness-side credential outage compensated by identical 10-minute makeup windows, and a shared session-limit stall; both runs' gate-fatal commits predate the stall (Section 11.8, incidents 1–2). [b]Gemini's final turns were foreclosed by a harness-side driver wedge held at round-4 parity; its judged regressions predate the wedge, which is therefore verdict-neutral (incident 3).

Contestant	Harness	Gate (O2/ASan regr.)	Hidden	Public	Board	R4 (hidden/board)
Fable 5-ultra†	workflow	PASS (0/0)	5.19x	5.08x	5.19	3.80 / 3.80
Opus 4.8	default	PASS (0/0)	3.41x	3.47x	3.41	4.20 / 4.20
Fable 5†	default	PASS (0/0)	3.15x	3.13x	3.15	3.94 / 3.94
GPT-5.5	codex	PASS (0/0)	1.27x	1.29x	1.27	1.38 / 1.38
Opus 4.8-ultra[a]	workflow	regressed (2/0)	4.56x	4.44x	0	4.79 / 4.79
Opus 4.7-ultra[a]	workflow	regressed (16/14)	4.23x	4.12x	0	4.33 / 0
Opus 4.7	default	regressed (16/16)	2.76x	2.78x	0	5.11 / 0
Gemini 3.5[b]	agy	regressed (7/7)	2.34x	2.29x	0	2.66 / 2.66

11.3 The A/B verdict: the behavior moved; the geomeans cannot carry the rest

The honest summary of the prompt A/B is asymmetric. The *behavioral* evidence is strong, replicated across both Fable 5 boxes, and maps point-for-point onto the failure modes the round-4 autopsy documented. The *score* evidence is two numbers pointing in opposite directions — Fable 5-ultra up 3.80→5.19, Fable 5 down 3.94→3.15 — inside a control field whose own swings were just as large. We report the behavior as the result and the geomeans as context.

What changed in the default run's shape. Round 4's Fable 5 spent 41 minutes changing engine code, launched its full gate at minute 58 and held *inline* for the 71-minute verdict, then froze for 78 no-op iterations against a deadline it had mis-anchored 56 minutes early. Round 5's Fable 5 committed performance work continuously from minute 6 to minute 114 of the window: twelve performance commits at a steady ten-minute cadence, roughly 109 minutes of first-to-last-commit engine mutation against round 4's 41, with the last engine commit at minute 115 of the window against round 4's minute 56. It launched the detached full gate at minute 120 with a pre-armed verdict-recorder helper, kept its turns deliberately idle of heavy CPU while the gate ran on the live binaries, and had the PASS verdict (182,560 case-runs, zero regressions) auto-committed into its status file in-window, 56 minutes before its deadline. Its deadline belief was correct to within two minutes (round 4: 56 minutes early).

A terminal hold did still occur, and the claim must be stated precisely: the last engine commit landed at minute 115 of 240, and run-log iterations 53–98 are freeze-verification no-ops, so roughly half the window again produced no engine change. But the hold's character inverted. It sat on a banked full-gate PASS rather than ahead of an unread verdict; its per-turn rationale — "any new code would ship un-full-gated — exactly how five of eight contestants zeroed out last time," repeated in near-identical form across some forty turns — was arithmetically true, since the observed ≈63-minute gate no longer fit in the 57 minutes remaining; and it was exactly the behavior the brief prescribes. "Freeze elimination" is the wrong frame; "the hold became gate-justified and the

Table 10: Round-5 per-workload hidden-suite speedups (median-of-5, interleaved, checksum-validated). The four gate-zeroed runs are below the rule.

Contestant	numeric	strings	propchurn	calls	regex	gcpressure	parse
Fable 5-ultra	9.70	3.61	8.80	9.22	3.91	4.59	1.99
Opus 4.8	5.31	2.67	6.59	3.99	2.11	3.90	1.77
Fable 5	3.90	2.37	5.26	3.61	2.63	3.18	2.08
GPT-5.5	1.22	1.38	1.20	1.29	1.34	1.42	1.08
Opus 4.8-ultra	8.71	4.43	7.64	5.31	3.26	3.96	2.02
Opus 4.7-ultra	6.58	4.28	7.08	5.23	3.33	3.96	1.77
Opus 4.7	3.70	2.27	4.80	3.05	1.82	3.10	1.74
Gemini 3.5	4.31	1.96	3.49	2.78	1.79	2.33	1.13

productive window doubled" is the defensible one — and even that carries the weaker-phantom-deadline and CLI-version confounds stated in the design.

What the workflow run did with the same prompt. Fable 5-ultra executed, under blind conditions, the strategy the round-4 analysis had concluded a hard no-regression speed race demands — proof decoupled from progress, scheduled so that neither starves the other. It went to the build system first, which no Fable 5 run had touched in round 4: a flags-and-PGO commit (`eb1e33a`, minute 64; `-O3`, full LTO, static linking, and a disclosed profile-guided build trained on the seven public workload classes, with the profile data committed and retrained after every source change — a lever no round-4 run had used) took its self-bench from 2.96x to 4.19x before most of its algorithmic work. Its harness fanned out three workflow dispatches with seven specialist subagents in per-workload git worktrees, each instructed that every fast path must bail to the generic path; merges were staged in an isolated tree behind three sampled gates before integration. And it ran *two* sequential full gates: gate #1 returned PASS at about minute 152 on a checkpoint tree that was then held as a validated revert target — a pre-tested script could restore the byte-identical gated checkpoint, binaries included, without a rebuild — after which it integrated its final optimization round (`8a7712e`, minute 153: a member-property inline cache, environment pooling, string fast paths) and put the integrated tree under gate #2, whose PASS was banked fourteen minutes before the deadline. It had even pre-committed a cutoff rule for the contingency that gate #1's verdict ran late. The content is notable too: the member-property inline cache is the architecture round 4's Opus 4.7 rode to 5.11x raw — and the architecture round 4's Fable 5 invented in overtime and then excluded from its own tree as foreign and ungated. One character note survives the strategy change: when its finalize script silently failed to archive a gate artifact, the run caught its own overclaim and committed a correction (`8de17b2`) restating the actual evidence chain.

Why the geomeans cannot carry the conclusion. Three decompositions strip most of the apparent paradox out of "the variant's default run got *slower*." The second and third are verified against the archived raw artifacts; the first is an operator-side rebuild disclosed below.

First, round 4's 3.94x was inflated: its scored tree included four performance commits landed past the window's end by the stray optimizer process (Section 10, incident 6). A controlled rebuild of the three trees benches round 4's at-freeze tree at 3.12x, round 4's scored tree at 4.03x (independently reproducing the autopsy's +0.9x overtime estimate), and round 5's scored tree at 3.75x. The rebuild used one compiler and one machine shared across all three trees — *not* the official timing box, so magnitudes are indicative and the direction is the point — with the committed public workloads, five interleaved repetitions, and checksums verified. On a like-for-like in-window basis, then,

the round-5 default run produced a *faster* tree than its round-4 self by roughly +0.6x. Disclosure: unlike every other number in this section, this rebuild is an operator-side measurement whose driver and raw timing output were not archived; it should be read as reported, not re-verifiable from the repository.

Second, the field moved with it: against Fable 5's −0.80 hidden, the Opus 4.8 control dropped a near-identical −0.78 (and nearly the same on the public leg, on byte-identical workload files), so the residual decline is indistinguishable from the round-wide downshift documented in Section 11.6. Third, in log terms the largest control swing in the field is twice the size of Fable 5-ultra's +1.39 treatment swing. The geomean cell of this A/B is, by itself, uninterpretable at $n = 1$; what replicated is the behavior.

What did not change. The residual trait evidence cuts against any strong prompt claim and is reported with the same weight. For the second consecutive contest, the default Fable 5 never touched the Makefile — no `-O3`, no LTO, no PGO — while its sibling's flags-only commit was worth 2.96x→4.19x on the self-bench; and it still spent roughly half its window holding a frozen tree, however rationally. And it was the workflow run, not the default one, that ended up holding the series record — the orchestration layer is observably where Fable 5's caution became throughput, whatever the prompt did or did not contribute. Finally, neither Fable 5 run shows any sign of having noticed the prompt change: searches across both runs' complete session archives find no meta-commentary about the system prompt or any out-of-character behavior; both runs are recognizably the same verification-saturated persona the earlier rounds document. The default box's first iteration was truncated by a transient model-availability error seven minutes in — the only such error in the round — but its session log shows work from the first minute and its first performance commit at minute 6, *before* the error, with resumption eleven seconds later. The effective time lost was nil.

11.4 The round-4 champion, zeroed on the blind rerun

Round 4's analysis of its winner ended on a warning: Opus 4.8-ultra's gold had been "earned at the bench, not at the gate" — continuous optimization behind sampled gates, the detached full gate never launched, a sampled PASS recorded as the verification of record. The blind rerun replayed that strategy almost beat for beat, and this time the dice came up wrong. The run's own wrap-up says it plainly: "`~/fullgate.log` was empty (full gate never launched), so I ran the sampled gate as instructed," and "the 8,000-case sampled PASS across both binaries is the verification of record" (in the gate kit's convention, a 4,000-case random draw run on both binaries — 8,000 case-runs). Its transcripts show nine sampled gate runs of 1,500–4,000 cases — and four separate occasions on which it *read* the detached full-gate launcher script without ever executing it, despite the brief's explicit instruction. The official gate found two regressed case-runs out of 91,280: `RGI_Emoji.js` (strict and sloppy modes), failing with "regular expression too complex" on the O2 binary only. Bisection pins both to a single commit (`81e7035`, minute 144 of its window) whose message opens "Integrated from parallel regexp.c work" and whose stated evidence is a 1,500-case sampled gate plus a 14-pattern differential test — the same integration-plus-sampled-gate class as round 4's autopsy of this run's near-misses. The mechanism is a resource-budget regression, deterministic and optimization-level dependent: moving the regex continuation chain from arena allocation to a small C-stack buffer pushes one enormous generated emoji alternation over the engine's complexity budget on the O2 build, while the ASan build's different frame layout stays under it. Against a two-case needle, the run's roughly 12,500 post-commit sampled case-draws had on the order of a one-in-four cumulative chance of ever seeing it.

The replication makes the survivorship structure of round 4's podium explicit. The same config-

uration, executing the same evidence policy in two statistically independent runnings, finished once as champion at 4.79x and once as a zero at 4.56x raw; nothing about its verification distinguished the two runs — only what the unexamined 95.6% of the suite (the 87,280 cases outside its final 4,000-case sample) happened to contain. Two corroborating details sharpen the point. The same `RGI_Emoji.js` pair is the case family that zeroed Opus 4.7-ultra's O2 leg in round 4; and in round 5 it was independently broken *again*, by the same stack-allocation idea, in Opus 4.7-ultra's own tree (commit `40b47bf`) — two different models converging on the identical optimization and the identical fatal case — while the Opus 4.8 default run flirted with the same trap and was saved by its own sampled gate catching a related regression, prompting a revert. (Round 4's Opus 4.8 had hit it too, and repaired it by heap-allocating instead.) A two-case, O2-only, deterministic needle that three distinct runs have now stepped on is a property of the optimization landscape, not of any one model; it stands as the series' canonical example of a regression sitting below practical sampled coverage.

11.5 0-for-6: the Opus 4.7 family's gate record replicates blind

Round 4 left the Claude Opus 4.7 family 0-for-4 on gated contests; round 5 makes it 0-for-6, under blind conditions, with regression counts of 26/28 and 128/128 (round 3), 9/9 and 7/5 (round 4), and now 16/16 and 16/14 — with top-half raw velocity at the bench in five of the six appearances (round 5's default run, at 2.76x raw and sixth of eight, is the exception). The replication also refines *what* the trait is, because round 5's 4.7 runs were not careless by the round-3 yardstick. Opus 4.7 gated heavily: 29 sampled-gate verdicts, five of them REGRESSED and acted on, and two detached full-gate launches — both of which died without a verdict (terminated mid-ASan-leg, killer unidentified in the logs). Its sixteen regressions trace to three mid-window commits (an integer-index array fast path responsible for six of the eight distinct files, including a trio that crashes outright under ASan; a dense-array path breaking a Proxy case; and an integer-modulus path losing −0 semantics). The damning sequence is narrow and precise: minutes into its wrap-up, its own randomized sample surfaced one manifestation of the array-fast-path family; it fixed exactly that case, re-ran a sample, recorded a PASS gate posture — "the strongest signal landed in-window" — and shipped, with six sibling manifestations of the same root cause still in the tree. Opus 4.7-ultra is the same story at higher discipline: it gate-fixed three of its own sampled-gate catches in-window — twice on the same regex first-character filter, whose root cause it never re-derived and which still ships broken — and its wrap executed a genuine save, rolling back four late inline-cache commits (self-benched at 4.64x) after its own 4,000-case sample caught 44 regressions, deliberately sacrificing roughly 0.6x of self-benched geomean for a clean sample. The judge still found 16/14 in what it kept. The family's failure mode, as of round 5, is not "does not verify"; it is *instance-fixing without root-cause closure*, at a regression-introduction rate that sampled coverage cannot police. Gemini 3.5's zero (7/7) is the adjacent lesson: its regressions bisect to its first commit and to its last — the latter landed after the last gate evidence of any kind, and its only completed full-gate verdict arrived, unactionably, after its final turn (Section 11.8). Notably, four of its regressed case-runs (the typed-array `concat` pairs) also appear in Opus 4.7's list — independent breaks in the same dense-array fast-path territory.

11.6 What the controls reveal: variance bounds every number in the series

Round 5's most consequential output may be its error bars. The replication provides, for the first time in the series, six same-configuration reruns with nothing intentionally changed — and their swings are large.

Measurement and suite noise are tightly bounded. The public leg was timed on byte-identical

files in both rounds; the public-versus-hidden overfit ratios deviate at most 2.7% from parity in either round; and regenerating the hidden suite contributes within about 2% for seven of eight runs (maximum 4.4%). Moreover, every control's public-leg movement is at least 92% as large as its hidden-leg movement, and in four of six controls *larger* — up to nearly double for Opus 4.8-ultra, consistent with its +4.4% suite component. That rules out the suites as the source: the swings are fully present on byte-identical files.

What remains is the agent. On identical workload files, the six controls all moved *down*, by raw deltas from −0.10 (Opus 4.7-ultra) to −2.35 (Opus 4.7, whose 5.11x raw collapsed to 2.76x — a 46% drop, the largest in the field). Opus 4.8 dropped 4.20→3.41 by shipping a visibly smaller subset of its own round-4 menu — 17 performance commits against 30, no LTO where round 4 had enabled it, an equivalent of one round-4 optimization parked on an unmerged branch explicitly labeled ungated, and a last engine commit 29 minutes *before* its deadline where round 4's landed 11 minutes after — buying a clean 0/0 PASS at the cost of roughly 19% of raw speed.

In log terms, computed throughout on the official hidden leg, the largest control swing ($|\ln| = 0.62$) is twice the Fable 5-ultra treatment effect (+0.31), and the Opus 4.8 control swing (−0.21) is the same size as the Fable 5 default's treatment swing (−0.23). That all six controls moved in the same direction is itself unexplained: a two-sided sign test gives $p \approx 0.03$, but the uniform CLI-version change cannot account for it (the GPT-5.5 and Gemini runs, on unchanged tooling, dropped too), so it is either chance or an unidentified round-level depressor. The retroactive implication is the point: every single-run comparison in this study — podium gaps, harness-versus-default deltas, model-versus-model orderings within a round — must be read against agent-level round-to-round variance of this magnitude. We flag in particular that round 4's 3.94-versus-3.80 Fable 5 ordering, already called provisional there, is well inside it.

11.7 Gate launches rose; banked verdicts stayed rare — and the zero rate doubled

The census requires a careful definition, because round 4's published "three of eight" (Section 10) counted full-suite *completions* before the official verdict, not launches. Recounted from round 4's own gate logs on a launch metric, at least six of the eight round-4 runs produced full-gate banners at some point (one launch was post-window, one came too late to finish, and the winner's own wrap says it never launched one); what was genuinely rare was an in-window *banked* verdict — two, both Fable 5 runs.

Round 5 moved the launch needle modestly but visibly: all six Wave A runs launched the detached full gate in-window, a unanimity round 4 never had, while Wave B launched none — and the banked-verdict count barely moved (two PASSes, both Fable 5 runs again, plus one completed verdict its owner could not read). Yet the zero rate doubled, from 2/8 to 4/8. The reconciliation is that launching a gate is not the win condition; *having a gate-verified tree, and time and ability to act on the verdict*, is.

Among the four zeroed runs, not one full-gate launch ever produced an actionable verdict: Opus 4.7's two launches died mid-run; Gemini's completed — REGRESSED — only after its final turn, behind a driver wedge, against mid-run binaries; and neither Wave B ultra launched one at all, both shipping on sampled evidence like round 4's champion.

Meanwhile both Fable 5 runs banked in-window full-gate PASS verdicts (round 4: two such verdicts field-wide; round 5: two, plus Gemini's unread REGRESSED), and Fable 5-ultra's endgame — two full gates plus a revert target — is, so far, the only observed pattern that solves the end-of-window dilemma completely: it ended the race holding both a gate-verified fast tree and a gate-verified fallback. The round-5 gate kit itself demonstrably works — randomized samples caught and

triggered five acted-on REGRESSED verdicts at Opus 4.7, a 44-regression catch and rollback at Opus 4.7-ultra, and a live save at Opus 4.8 — so the residual failure mode has moved down a level: needle regressions below sampled coverage, and instance fixes without root-cause closure. On-box full-gate verdicts also remained fallible (round 4: GPT-5.5's spurious REGRESSED-17,683; round 5: Gemini's mid-run REGRESSED-17,298 against binaries that no longer existed at collection, versus the judge's authoritative 7); judge-side gating remains the only ground truth in this series.

11.8 Operational incidents and threats to validity

Thirteen in-round incidents are recorded in the round's data file (`results/official/nitro2-data.json`) and reproduced, with two pre-launch disclosures, in Appendix B (incidents 24–38); the material ones follow. (1) A harness-side credential rotation broke authentication on both Wave B boxes for about 7 minutes early in the window; both were granted identical 10-minute makeup windows after their deadline. (2) Both Wave B runs later hit a shared-subscription session limit and idled until the limit reset, roughly half an hour before the deadline (roughly 15 and 24 minutes of logged bounce iterations for Opus 4.8-ultra and Opus 4.7-ultra respectively, with the former's commit gap suggesting a longer effective stall); both resumed for a final working turn, and — relevant to causal reads — every commit implicated in either ultra's gate verdict predates the session-limit stall, which was therefore contributory time pressure, not a cause. (3) The agy driver captures turns via command substitution, and Gemini's detached full gate inherited that pipe descriptor: when the CLI process exited, with roughly 45 minutes of window left, the driver never returned, costing Gemini its remaining turns (the incident log estimates ∼25 minutes; a log-based reading supports up to ∼45). The driver code is identical to round 4's (parity preserved), and the run's own completed gate verdict landed behind the wedge, after its last turn. (4) Opus 4.8's detached full gate was silently reaped by that box's sandbox the moment the launching tool call returned — an environment quirk; the run shipped on layered sampled gates and the official gate confirmed 0/0. (5) Opus 4.7's detached full gate was SIGTERM'd mid-ASan-leg by an unidentified killer. (6) The post-window-commit disclosure, uniform under the standing final-`main`-at-collection rule: Opus 4.7 landed three post-deadline engine commits in its wrap (two fixes and a `-O3 -flto` change, which its own status mislabels as in-window; bisection confirms they neither caused nor fixed any judged regression), GPT-5.5's final in-flight turn committed past its deadline, Opus 4.7-ultra's post-deadline action was the rollback described above (it subtracted code), Opus 4.8-ultra's post-makeup commits touch only documentation and tooling, and Gemini's dirty tree received a collector snapshot that is verdict-neutral (its last authored commit fails the judged cases identically). (7) The Fable 5 launch-time model error cost effectively nothing (Section 11.3). (8) Two pre-launch items are disclosed in the same spirit as the series' other caught-before-effect near-misses: the blindness audit initially *found* round-4 tells on the re-imaged boxes, which were fixed before launch was released; and the GPT-5.5 box, deleted after round 4, was rebuilt with its repository sourced from a sibling contestant box's pristine baseline preparation (its bundle's baseline tag verified identical to every other run's). All eight contestant boxes were deleted only after verified collection; bundles, run logs, gate logs, and full session archives are under `results/n2-*/` and `results/archive-nitro-r2/`.

The threats to validity are the design's confounds, restated as limits on inference. The CLI-version change (2.1.165 vs. 2.1.161) and the weaker phantom-deadline structure both sit inside the Fable 5 A/B, so even the behavioral deltas — the doubled productive window, the correctly-anchored gate-justified hold — are attributable only to "the round-5 condition," not to the prompt specifically; we claim replication-grade evidence that the round-4 pathology *did not recur*, not proof of why. The makeup windows and session-limit stalls make Wave B's effective budgets unequal to

Wave A's in ways that were equalized within Wave B but not across waves (a structure round 4 shared). The Gemini wedge means one control's final-turn capacity was harness-limited, though its gate verdict was already determined by commits that predate the wedge. And the variance finding cuts both ways: it is established on six controls at $n = 1$ each, so the round-level downshift it documents is itself only bounded, not explained. None of these threats touch the round's internal board: within round 5, all eight runs faced identical suites, identical gates, and a judge-side verdict on the tree each one shipped.

References

[1] Anthropic. Claude code. `https://claude.com/claude-code`, 2026.

[2] Fabrice Bellard. QuickJS javascript engine. `https://bellard.org/quickjs/`, 2024.

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

[4] Containers project. bubblewrap: Unprivileged sandboxing tool. `https://github.com/containers/bubblewrap`, 2025.

[5] Ecma International. ECMA-262: Ecmascript 2025 language specification. `https://tc39.es/ecma262/`, 2025.

[6] Ecma International, TC39. test262 INTERPRETING.md: How to interpret and run the tests. `https://github.com/tc39/test262/blob/main/INTERPRETING.md`, 2026.

[7] Ecma International, TC39. Test262: Ecmascript test suite. `https://github.com/tc39/test262`, 2026. Corpus pinned at commit 4249661388e5d3f92a85186213da140a6481490f.

[8] 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.

[9] QuickJS-ng contributors. quickjs-ng: A fork of QuickJS. `https://github.com/quickjs-ng/quickjs`, 2026. Version 0.15.0.

[10] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Address-Sanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, 2012.

A Contestant Prompt and Driver Prompts

Every contestant received the same contest brief (`BRIEF.md`), staged on its box at `~/js262-faceoff/BRIEF.md`. Runs were driven by a shell loop in a `tmux` session under an unprivileged `contestant` user: each turn invoked the agent CLI non-interactively with a per-turn timeout of 2,700 s, and the loop re-invoked the CLI until the wall-clock budget expired. For Claude-CLI contestants the first turn received the full brief and later turns either resumed the transcript (`--continue`) or, in FRESH mode, started a new session each turn that re-oriented from on-disk state (FRESH mode was introduced after a 32 MB transcript-resume limit wedged a long session; see Appendix B, incident 2). The `codex` contestant (GPT-5.5) received the full brief on its first turn and resumed

its own session (`codex exec resume --last`) with a short continuation prompt thereafter. The max-effort "ultra" variants additionally ran with the CLI's maximum effort setting and had a workflow directive appended to every prompt. This appendix reproduces the brief in condensed but faithful form, followed by the driver prompts verbatim.

A.1 The contest brief (condensed)

Mission. "Implement a JavaScript (ECMAScript) engine in C, from scratch, in a single session. Your score is the number of official tc39/test262 conformance cases your engine passes under the judge's reference runner. You will not pass them all. Pass as many as possible."

Hard rules (reproduced near-verbatim):

1. *Language:* C (C11 or C17). Dependencies: libc and libm only. No third-party libraries, no code generators, no garbage-collector libraries, no bignum libraries.

2. *From scratch:* the contestant may not read, copy, port, or adapt source code from any existing JavaScript engine or JavaScript parser (QuickJS, JerryScript, duktape, mujs, V8, JavaScriptCore, SpiderMonkey, engine262, Babel, swc, etc.), and may not invoke an existing JS engine at build time or runtime.

3. *Allowed references:* the ECMA-262 specification, the test262 repository (tests, harness files, INTERPRETING.md), MDN, and the documents in the contest repo.

4. *No test-gaming:* no logic keyed to specific test262 file names, paths, or recognizable test contents. Hardcoding spec constants (e.g. `Number.MAX_SAFE_INTEGER`) is fine; hardcoding answers to tests is not. The brief notes that the official runner strips test metadata and randomizes filenames, and that the session transcript will be audited.

5. *Single session, no human assistance* after the prompt. Budget: 8 hours of wall-clock time from the moment the prompt is submitted, identical for every contestant. When the budget expires, work stops wherever it is; the final committed state of `main` in `~/js262-submission` is what gets scored.

Enforcement. The brief states explicitly how the rules are enforced: the judge builds with `clang` and records warnings; runs `ldd`/`readelf` on the binaries and flags any shared library outside libc/libm/libasan/libubsan and the C runtime; runs hidden anti-cheat probes whose required output is a *computed* value ("an engine that doesn't actually execute JavaScript fails them"); and reviews the full session transcript for rule 2 / rule 4 violations. A high public score combined with failed probes, forbidden dependencies, or transcript evidence of copied source is treated as cheating and disqualifies the submission.

Environment. A dedicated Ubuntu 24.04 x86_64 box (4 vCPU / 8 GB) with `clang` (default compiler), `gcc`, `make`, `git`, `python3`, `uv`; test262 pre-fetched at the pinned SHA; the reference runner (`runner/run262.py`) available for self-measurement, with official scoring performed by the judge on a separate clean machine.

Deliverable. A git repository at `~/js262-submission` containing source and a `Makefile` with two targets: `make` producing `./js262` (`-O2`) and `make asan` producing `./js262-asan` (`-fsanitize=address,undefined -fno-sanitize-recover=all`); both must build cleanly under `-Wall -Wextra` with no warnings; plus a short `STATUS.md`.

Engine contract (condensed from the brief):

- Invocation: `./js262 [--module] file1.js ... fileN.js`. All files are evaluated in order in a single shared realm; files 1..$N-1$ are classic global scripts (the runner-supplied test262 harness); the last file is the test, evaluated as a classic script or, with `--module`, as an ECMAScript module (fixture specifiers resolve relative to the working directory).
- Exit code 0 = the test ran to completion with no uncaught exception. Nonzero exit = parse error or uncaught exception; the first non-empty stderr line must begin with the error constructor name followed by a word boundary (e.g. `SyntaxError: unexpected token`). Termination by signal always counts as a failure, even for tests that expect an error.
- A global `print(value)` builtin is required. The `$262` host object is optional, but implementing `createRealm`, `evalScript`, `detachArrayBuffer`, `gc`, and `global` unlocks roughly 600+ additional cases; `$262.agent` is not expected.
- Each case runs in its own temp directory with a cleaned environment; no reliance on cwd contents, environment variables, or network. Timeouts: 3 s per case, retried once at 10 s; persistent timeout = fail.

Scoring (condensed): the corpus is test262 at the pinned SHA, all `*.js` under `test/language/` and `test/built-ins/` excluding fixtures, `intl402/`, `annexB/`, and `staging/`; each test runs in strict and non-strict mode unless flagged otherwise, each mode counting as one case, with the exact case list fixed by the committed manifest. Pass criteria: a normal test passes on exit 0 with the completion sentinel printed; a `negative:` test passes on nonzero non-signal exit with the matching stderr error name; an `async` test passes on exit 0 with `Test262:AsyncTestComplete` on stdout and no failure marker; any ASan/UBSan report fails that case in the sanitizer run. The completion sentinel is described in the brief: the runner appends `print("Test262Sentinel:<token>")` with a random per-case token to every normal staged test and requires the token on stdout; extracting the token from source text without executing it is test-gaming under rule 4. The scoreboard defines S1 (cases passed, `-O2`), S2 (cases passed with the ASan/UBSan build at 3× timeout allowance, the primary ranking metric), S3 (the S1/S2 diff report), and S4 (total `-O2` suite wall time, tiebreaker for exact S2 ties only); ranking is S2, then S1, then S4.

Strategy guidance (non-binding, condensed): the brief notes the corpus is dominated by core language semantics and everyday builtins; that `propertyHelper.js` gates roughly 4,800 cases; that about 5,500 cases are async and require a job/microtask queue; and that about 4,600 are negative syntax tests earned by a strict, correct parser — "Spend your budget where the tests are."

A.2 Driver prompts (verbatim)

FRESH-mode continuation prompt (the per-turn prompt for FRESH-mode Claude runs):

```
Read ~/js262-faceoff/BRIEF.md for the task and scoring rules. Work in ~/js262-submission.
If a JavaScript engine already exists there, it is your own continuing work
--- do NOT start over; read STATUS.md and the existing source, run 'make',
and keep improving test262 conformance. If the directory is empty (no source
yet), begin building the engine from scratch per BRIEF.md. To self-measure
use a SMALL light sample to avoid running the box out of memory: 'python3
~/js262-faceoff/runner/run262.py --engine ./js262 --test262 ~/js262-faceoff/vendor/test262
--manifest ~/js262-faceoff/manifest.json --sample 500 --jobs 1' (do NOT use
large --sample or high --jobs; this 8GB box will OOM). Commit with git frequently.
```

Transcript-mode continuation prompt (non-FRESH Claude turns after the first, and `codex` resume turns):

```
Continue building the JavaScript engine in ~/js262-submission. Maximize the
number of test262 cases that pass (use ~/js262-faceoff/runner/run262.py to
measure yourself). Commit progress with git frequently. Do NOT stop or declare
the task done --- keep improving until the session ends.
```

Workflow directive (appended to every prompt for the ultra runs, i.e. Opus 4.8-ultra and Fable 5-ultra):

```
IMPORTANT: use a workflow --- spawn parallel sub-agents to build and improve
the engine's subsystems concurrently (lexer, parser, interpreter, property
descriptors, builtins, RegExp, TypedArrays, Promise/async), then integrate and
re-measure. Keep launching workflows to attack remaining test262 gaps.
```

A.3 Wrap-up prompt (verbatim)

When a contestant's main run ended, a separate one-hour wrap-up session was launched with the following prompt:

```
You have ONE HOUR of clock time to wrap up your JavaScript engine in ~/js262-submission
and get it ready to test. First ensure BOTH make and make asan build cleanly
under -Wall -Wextra with zero warnings (run them to confirm). Then finalize:
fix regressions and improve test262 conformance where low-risk. SCORING
REMINDER: you are evaluated on the ENTIRE test262 suite, reported as TWO
separate buckets --- (1) NEGATIVE tests (must throw an uncaught error of the
expected type, nonzero exit) and (2) POSITIVE tests (must run to completion
and pass). Make sure BOTH buckets are strong; do not neglect either (e.g. a
strict, correct parser earns the large negative/SyntaxError bucket, while real
execution earns positives). Measure with python3 ~/js262-faceoff/runner/run262.py
--engine ./js262 --test262 ~/js262-faceoff/vendor/test262 --manifest ~/js262-faceoff/manifest.json
--sample 500 --jobs 1. Commit frequently; the final committed state of main is
scored.
```

Subsequent wrap-up turns within the hour resumed the session with the short prompt "Continue wrapping up: keep make and make asan building cleanly, finalize, commit."

B Incidents and Fairness Rulings

The contest was run live on real infrastructure, and several operational incidents occurred. Each is documented here together with its resolution and, where applicable, the fairness ruling.

1. *API-credit exhaustion (Opus 4.7, Opus 4.8 originals).* The original Claude Opus 4.7 and Opus 4.8 runs were degraded by operator-side API-credit exhaustion roughly 48 minutes in. The boxes' credentials were remediated and both contestants were granted clean 8h+1h fair reruns. The degraded artifacts were preserved (`results/*-degraded-run.bundle`, `*.degraded.log`), not discarded.

2. *Transcript-resume limit.* Claude Code's 32 MB `--continue` resume limit wedged a long session. The driver was changed to FRESH mode: a fresh session per turn, with state re-read from disk rather than carried in the transcript (Appendix A).

3. *Contestant-induced memory pressure.* Contestants' own unthrottled debug runs ballooned memory (observed up to 7 GB) and OOM'd boxes. A 4 GB swap file and a periodic process reaper (age/RSS thresholds) were added.

4. *Account session-limit contention.* Running multiple concurrent Opus contestants on a shared subscription account hit session limits; concurrent Opus runs were serialized to at most two.

5. *Gemini 3.5 Flash run.* The agy-driven Gemini run hit quota exhaustion and then a credential expiry that could not be refreshed headlessly; the run was parked early and is reported as incomplete. It also ran on the operator's Mac (the only practical agy host) rather than a Hetzner box — a hardware deviation — and was scored on the Mac without the bubblewrap sandbox (a documented deviation).

6. *Premature auto-scoring.* A `launchd` auto-scoring job fired prematurely and produced an empty scoreboard. The job was disarmed; all official scoring was performed manually.

7. *ASan probe-pass harness bug.* The first scoring pass false-tripped the cheat flag on every engine: the probe pass ran the ASan binary without sanitizer-mode runner settings, so `RLIMIT_AS` killed ASan's shadow-memory reservation at startup (0/40 ASan probes, with an identical error for all engines). This was diagnosed as a harness bug; `score.sh` was fixed (the ASan probe pass now uses `--sanitizer` and the 3× timeout), and all engines were re-scored. S2 was identical case-for-case for every contestant, probes returned 40/40 on both binaries for every engine except GPT-5.5 (37/40 on both binaries, genuine computed-output errors), and the flags were cleared.

8. *Branch-name technicality (Opus 4.7).* Opus 4.7's submission repository had only a `master` branch and no `main`. Ruling: the submission was staged with `main` created at `master`'s head rather than zero-scoring the run on a branch-name technicality.

9. *Fable 5 wrap-up overrun.* The non-ultra wrap-up script lacked the main driver's per-turn timeout, so a single turn spanning the one-hour deadline ran unbounded; Fable 5's wrap-up consequently overran to roughly 2 hours before the operator cut it. Other contestants' wrap-ups ran roughly 1–1.5 hours under the same script (one was operator-cut at 51 minutes). Ruling: documented as a harness gap, not a contestant violation; the operator cut restored approximate parity, and the final commit landed during the overrun window was spec-minutiae polish (Date-setter and RegExp group-name details), so the score effect is judged small. The residual wrap-up-time inequality is acknowledged in Section 7.

10. *Wave B session-limit stall (round 3).* Both round-3 Wave B runs (Opus 4.8-ultra, Opus 4.7-ultra) were stalled by a shared subscription session limit — an approximately 88-minute mid-run stall. Ruling: makeup windows were granted sized to the verified stall time (162 minutes for Opus 4.8-ultra, 88 minutes for Opus 4.7-ultra).

11. *Gemini 3.5 quota throttling (round 3).* The round-3 Gemini 3.5 run was quota-throttled by HTTP 429 storms (with roughly 5-minute reset cycles) from about three hours in; its authentication held, a watchdog false-flagged the run once (cleared), and it finished its full window. No time adjustment was made.

12. *Judge-side scoring wedge (round 3).* A judge-side scoring session wedged in its terminal multiplexer and was relaunched, with no effect on results.

13. *Replay-pipeline fixes (round 3).* The milestone replay pipeline needed two fixes — v1 scored under the sandbox wrapper while the contest ran bare, and v2 hit checkout conflicts on tracked binaries, resolved with a force-checkout — after which all replays were rerun. All round-3 crossing times reported in Section 9 are from the rerun.

14. *Gate-queue v1 invalid (round 4, judge-side).* The first judge-side gate-queue launch used a relative `--exec-wrapper` path, so all 91,280 cases errored at the runner on every leg and every run showed the same impossible count of roughly 90,436 "regressions." A sanity check tripped on the absurd identical numbers before any verdict was recorded; the path was absolutized and the queue fully relaunched. The second occurrence of this bug class in the series' judge tooling (cf. the round-3 flaky-list queue); no contestant artifact or verdict was affected.

15. *Timing-box package index (round 4).* The freshly provisioned dedicated-core timing box had an unpopulated apt index and engine builds failed until it was updated; resolved before any official timing ran.

16. *Opus 4.7 post-window "GATE FIX" (round 4).* Opus 4.7's head commit landed roughly 50 minutes past its window end, about 20 minutes into its wrap session. Ruling: the contract scores the final committed `main` at collection, so the late commit was part of the gated tree — which still regressed 9/9. The verdict stands on the tree the contestant shipped; whether the window-end tree alone would also have regressed was not separately tested.

17. *Opus 4.7-ultra session limit (round 4).* Hit the shared-subscription session limit at iteration 6, roughly 15 minutes before its window end; iterations 7–10 bounced and iteration 11 started with roughly 11 minutes of window left. Zeroed on the gate regardless.

18. *Wave B late wrap-ups (round 4).* Both Wave B wrap sessions fired roughly 10.8 hours after their windows ended (a scheduling fault). Opus 4.8-ultra's wrap discovered its detached full gate had never been launched, ran a final 8,000-case sampled gate, and recorded that PASS explicitly as partial evidence; Opus 4.7-ultra's wrap produced no captured log, though its tree gained two engine fixes and a rebuild minutes before collection (incident 22).

19. *Fable 5 stray-process edits (round 4).* A still-live optimizer process from Fable 5's final in-window iteration committed four post-window performance commits (in the scored tree under the collection rule, covered by the official 0/0 gate) and later left uncommitted edits in the working tree; the contestant's own wrap-up stashed the uncommitted batch as foreign/ungated, proved the clean-HEAD rebuild bit-identical to the gated binary, and re-gated. Only the uncommitted batch is excluded from the scored tree.

20. *Collector snapshot commits (round 4).* The Gemini 3.5 and Opus 4.7-ultra trees were dirty at collection; the collector committed "final state" snapshots, and contestant-authored commit counts exclude that snapshot.

21. *Opus 4.7-ultra gate legs last to finish (round 4, judge-side).* Queue order put its legs last; the ASan leg completed roughly 18 hours after the Wave B window ended, with final counts 7 O2 / 5 ASan, the ASan regression set a strict subset of the O2 set.

22. *Post-window commits field-wide (round 4).* Collection occurred roughly 17 hours (Wave A) and 11.7 hours (Wave B) after the windows ended, so the final-`main`-at-collection rule admitted post-window work broadly; post-window engine commits exist in five of the eight scored trees (Opus 4.7, Fable 5, Opus 4.8, Opus 4.8-ultra, Opus 4.7-ultra) and were disclosed and ruled on identically for every run (Section 10).

23. *Public-leg suite regeneration (round 4, judge-side).* The first official public-suite timing leg accidentally benched a fresh seed-1000 regeneration rather than the committed development files, because the workload generator's per-class streams pass through Python's per-process

salted `hash()` and are not byte-stable across invocations. Caught in audit; the public leg was re-timed on the literal committed files with the same binaries, box, and methodology, the published Public/Overfit figures use the committed-suite leg, the two legs agree within noise, and the hidden-suite board was never affected. All scored suites, drivers, and raw timing logs are archived under `results/official/`.

24. *Pre-launch blindness audit finds tells (round 5, pre-launch).* Before the blind replication launched, a dedicated audit swept every re-imaged box for round-4 residue and rerun tells; it initially *found* blindness blockers, which were fixed, and launch was held until the audit returned clean. Post-hoc transcript greps found zero references to any prior run on any box; brief parity was separately verified (the brief's "five of eight shipped sampled-PASS regressions" warning describes round 3 and appears verbatim in both rounds' logs).

25. *Rebuilt GPT-5.5 box, sibling-sourced (round 5, pre-launch).* The round-4 GPT-5.5 box had been deleted; its round-5 box was newly created (an unrelated box was deleted, after a backup snapshot, to free the server-limit slot), with the contest repository sourced from a sibling contestant box's pristine baseline preparation rather than any round-4 artifact. Its collected bundle's `baseline` tag matches every other run's.

26. *Credential-rotation 401 outage (round 5, Wave B, harness-side).* A credential rotation broke authentication on both Wave B boxes for about 7 minutes (minutes 18–26 of the window); both logged instant-401 bounce iterations. Ruling: identical 10-minute makeup windows for both after their deadline. The outage predates none of either run's gate-fatal commits, which all landed later (and before the session-limit stall, incident 27).

27. *Wave B session-limit stall (round 5).* Both Wave B runs hit the shared subscription session limit late-window and idled until its reset roughly 33 minutes before the window end (roughly 24 and 15 minutes of bounce iterations; Opus 4.8-ultra's commit gap suggests a longer effective stall). Both resumed for a final working turn; every commit implicated in either run's gate verdict predates the stall — contributory time pressure, not a cause.

28. *agy driver descriptor wedge (round 5, Gemini; parity ruling).* The agy driver captures turns via command substitution, and Gemini's detached full gate inherited that pipe descriptor: when the CLI exited roughly 45 minutes before the window end the driver never returned, foreclosing the run's remaining turns ($\sim$25 minutes conservatively; up to $\sim$45 by the strictest reading). The driver code is byte-identical to round 4's (parity preserved); discovered at collection, so no makeup was possible. The run's own completed gate verdict landed behind the wedge, after its last turn, unactionably.

29. *Gemini on-box gate verdict vs. judge (round 5).* Gemini's completed full gate reported REGRESSED-17,298 against mid-run binaries; the authoritative judge-side gate on the final tree found 7. Both numbers preserved; on-box full-gate verdicts were unreliable in 2 of 16 legs across the two nitro rounds, and judge-side gating remains the only ground truth.

30. *Opus 4.8 sandbox reaps detached gates (round 5).* That box's sandbox reaped detached background processes the moment the launching tool call returned, so its in-window full-gate launch silently died (root-caused by the run's own wrap). It shipped on layered sampled gates; the official gate confirmed 0/0. Box-specific quirk; both Fable 5 boxes' detached gates completed normally.

31. *Opus 4.7 full gate SIGTERM'd (round 5).* Both of its in-window full-gate launches died mid-ASan-leg without a verdict (killer unidentified in the captured logs, plausibly its own rebuild/cleanup tooling); it shipped on a sample-1500 PASS and was zeroed 16/16.

32. *Opus 4.7-ultra makeup-window rollback (round 5).* Its wrap rolled back makeup-window inline-cache work (self-benched 4.64x) after its own 4,000-case sample caught 44 regressions — a genuine post-deadline save that subtracted code — and the kept in-window tree still carried 16/14.

33. *Fable 5 launch model error (round 5; corrected record).* Iteration 1 errored with a transient model-availability message roughly seven minutes into the window — after real work from the window's opening seconds and the run's first performance commit at roughly six minutes in — and iteration 2 resumed eleven seconds later. Effective loss approximately nil (an earlier note of "∼7.7 minutes lost" read the iteration boundary as dead time and is corrected in the data file).

34. *Missing wrap logs (round 5).* No wrap-up session logs exist for Gemini (driver wedged until collection, incident 28) or GPT-5.5 (its final turn ran past the deadline and no separate wrap session was produced); both boxes' scored trees, run logs, and gate logs were collected normally.

35. *Post-window commits, field-wide disclosure (round 5).* The final-`main`-at-collection rule again admitted post-window work uniformly: three post-deadline engine commits at Opus 4.7 (including a timing-relevant `-O3 -flto` change; bisection confirms they neither caused nor fixed any judged regression), three at GPT-5.5 from its in-flight final turn (gated 0/0), Gemini's collector snapshot (verdict-neutral), and Opus 4.7-ultra's rollback pair. The two Fable 5 runs and both Opus 4.8 runs landed none.

36. *Collector snapshot commit (round 5).* Only Gemini's tree was dirty at collection; the collector committed a "final state" snapshot, excluded from its contestant-authored commit count and verdict-neutral (the last authored commit fails the judged cases identically).

37. *Infrastructure reuse and teardown (round 5).* The judge and timing boxes were reused from round 4 under a strict separate namespace, leaving all round-4 records untouched; all eight contestant boxes were deleted only after verified collection, with bundles, logs, and full session archives preserved in the repository.

38. *codex continue-prompt parity (rounds 4–5, GPT-5.5).* The codex driver's continuation prompt carries the series' generic conformance wording rather than nitro speed wording in *both* rounds (five occurrences in each run log) — a driver quirk held at parity, not a round-5 delta; the nitro brief was delivered in iteration 1 both times and both runs did speed work throughout.

Fairness principle. Every run materially degraded by operator or harness error received a clean rerun or a clean resume; degraded artifacts were preserved and documented, not silently discarded. Residual inequality in effective wall-clock time across contestants nevertheless remains and is acknowledged in Section 7. Table 11 consolidates the official results with each run's integrity record and final status after all fairness rulings.

Table 11: Consolidated results reference. Official conformance (S2 = sanitizer-clean passes of 91,280, the primary metric; S1 = O2 passes), secret computed-output probes (identical on the O2 and sanitizer binaries for every run), audited effort metrics, and run status after all fairness rulings. Canonical three above the first rule, ordered by S2; the Fable 5 row is the canonical-protocol addendum (run after the v1 draft, identical protocol and pins); add-ons below the second rule are not rank-comparable. Superscripts refer to the incident list above; Gemini 3.5's tree was not audited (incomplete run).

	Conformance			Probes	Effort			
Run	S2	S2 (%)	S1 (%)	passed	Commits	LOC	Files	Status
Opus 4.8	81,160	88.91	88.95	40/40	103	19,901	31	fair rerun[a]
Opus 4.7	59,439	65.12	65.30	40/40	149	15,683	2	fair rerun[a,b]
GPT-5.5	41,702	45.69	45.99	37/40	282	10,777	1	clean
Fable 5	89,191	97.71	97.72	40/40	37	30,043	27	clean; wrap operator-cut[c]
Fable 5-ultra	88,347	96.79	96.87	40/40	81	43,599	60	clean
Opus 4.8-ultra	84,740	92.84	92.93	40/40	60	33,221	43	clean
Gemini 3.5	29,549	32.37	32.72	40/40	139	–	–	incomplete[d]

[a] Original run degraded by operator-side API-credit exhaustion; clean full-budget rerun scored (incident 1). [b] Staged with `main` created at `master`'s head (incident 8). [c] Clean run; its 1 h wrap-up overran to roughly 2 h because the wrap script lacked a per-turn timeout, and was operator-cut (incident 9). [d] Parked early after quota exhaustion and a non-refreshable credential expiry; ran and was scored on the operator's Mac without the sandbox (incident 5).

C Run Timelines

The capsule timelines below are mined from each submission's git log (`results/metrics/*.gitlog`); times are given as elapsed time from each run's first commit. Commit-time spans do not equal active budget: they include incident-related pauses, blocked or hung turns, serialized account access, and operator-granted resumes documented in Appendix B; residual unequal effective wall-clock is discussed in Section 7. Commit counts are from the verified metrics in Section 5.

Opus 4.8 (canonical, clean rerun, 103 commits). First commit "Initial from-scratch JS engine: parser, interpreter, core builtins"; Promise plus the async/await driver 15 minutes later, Proxy with full trap dispatch within the first hour, a real regex engine at roughly 1.5 hours, the `$262` host object at roughly 3.2 hours, and Temporal (namespace + PlainDate) starting at roughly 3.5 hours. The ES module system (loader, linking, live bindings, namespace exotic object, dynamic `import()`) landed late, at roughly 7.2 hours. The run closed with parser early-error and small-builtin commits, the last (`42763df`, lexer ZWNJ/ZWJ identifier handling) at roughly ten hours; for this run the mined gitlog ends early (94 commits, stopping roughly half an hour before the final commit), so the closing commits are taken from the scored bundle (103 commits).

Opus 4.7 (canonical, clean rerun, 149 commits). First commit "WIP: lexer + parser skeleton"; a basic engine with `assert.js` compatibility within the first half hour, and Map/Set/Promise/Date stubs plus the `$262` host minutes after that. The middle of the run followed a visible stub-then-refine pattern across builtins (RegExp/ArrayBuffer stubs at roughly 40 minutes, modules and top-level await at roughly 2.9 hours, DataView / TypedArray prototype work mid-run), peaking at 28 commits per hour around hours three through five. Final commits, including completion-value semantics for loops, ran to roughly ten hours after the first commit. The submission remained a two-file monolith.

GPT-5.5 (canonical, 282 commits). First commit "Implement initial C JavaScript interpreter", then a strikingly steady cadence of small commits (roughly 15–20 per hour for some sixteen hours): RegExp constructor state at the one-hour mark, a Temporal stub already at roughly 1.3 hours, Proxy forwarding at roughly 1.7 hours. Main-run commits ended roughly sixteen hours after the first; after a gap of several hours, a final block of roughly an hour and a half opened with a "final-snapshot" commit and continued with generator and class-syntax fixes. The submission was a single-file monolith.

Opus 4.8-ultra (add-on, 60 commits). The first commit was already a working baseline engine (core language plus essential builtins), consistent with large integration commits from subagent workflows (14+ agent/workflow worktrees were observed). The `$262` host landed within the first hour, a parser early-errors pass at roughly 3.2 hours, the full ES module system together with DisposableStack/AsyncDisposableStack at roughly 7.4 hours, Atomics workflow preparation at roughly 10.4 hours, and `BigInt.asIntN/asUintN` at roughly 13 hours. The log ends with reserved-word and `new.target` early-error fixes and a terse "final" commit at roughly 14.3 hours.

Gemini 3.5 Flash (add-on, incomplete, 139 commits). Run on the operator's Mac under agy (incident 5). First commit "checkpoint: pre-restart state", followed by 53 commits in the first hour (array methods, `this` binding, object reflection). Activity then came in bursts (the first two hours, a stretch around hours nine to ten, and a final stretch through roughly hour fifteen) separated by quota-exhaustion and credential-expiry outages; a STATUS.md self-score of 189/500 on a sample was committed roughly ten hours in. The last commit, roughly 15.7 hours after the first (BigInt lexer/parser work), is where the run was parked.

Fable 5 (canonical-protocol addendum, 37 commits). A working base engine landed first, followed by the major builtins, then ES modules with top-level await (built on fibers), SharedArrayBuffer and Atomics, BigInt, explicit resource management (`using` / `await using`), and iterator helpers; the `$262` host object was implemented early. Mid-run the contestant renamed its own `master` branch to `main`, citing the scored-branch rule (cf. incident 8). A self-measured 84% at roughly the five-hour mark rose to 492/500 (98.4%) in the wrap-up. The final commits, landing during the wrap-up overrun window (incident 9), were spec minutiae: Date-setter edge cases and RegExp group-name details.

Fable 5-ultra (add-on, 81 commits). The first commit opened architecture-first, with a skeleton (arena, strings, values, module contracts) followed by a stub scaffold of the entire engine behind feature flags; a `STATUS.md` laying out the design landed roughly an hour in. Parallel workflow agents landed whole modules mid-run (commits such as "interp_expr.c + interp_stmt.c land (workflow agents)"), with the stub flags dropped one by one as modules became real. An integration surge of 18 commits landed in the final hour; ASan stack-size tuning occurred during the wrap-up, which ended naturally within its hour. At 43,599 audited lines across 60 files, it submitted the largest engine of the study.