E2E-SWE: BENCHMARKING LLMS ON BUILDING WORKING CODEBASES FROM SCRATCH

Anonymous authors
Paper under double-blind review

ABSTRACT

Coding agents powered by large language models (LLMs) are evolving from making localized code changes to developing complete software repositories. However, evaluating repository-scale generation remains challenging: tasks must demand system-level reasoning while ensuring that all evaluated behaviors are precisely specified and independent of any particular implementation. We introduce E2E-SWE, a benchmark for evaluating whether coding agents can build complete, functional software repositories end to end. E2E-SWE contains 186 whole-repository generation tasks spanning 11 programming languages. Given only a natural-language specification and an empty workspace, an agent must implement a complete, installable project that satisfies a comprehensive suite of hidden tests. Each task is constructed by a software engineer in collaboration with an LLM; together, they develop the test suite and a corresponding implementation-independent specification. To ensure that tasks are well specified and practically solvable, we further subject them to an iterative verification process in which autonomous agents audit and repair task defects using static inspection and failures observed from real model rollouts. Evaluating 13 frontier models, we find substantial variation in end-to-end repository generation ability, with pass@1 ranging from 11.7% to 67.7%, providing strong model differentiation while leaving considerable headroom for future progress. Analysis of agent trajectories further reveals long, front-loaded reasoning patterns, highlighting the planning and system-level reasoning required to construct working codebases from scratch.

1 INTRODUCTION

Large language models (LLMs) have moved to the center of modern software engineering. In only a few years, LLM-powered tools have progressed from inline completion (Chen et al., 2021; Murali et al., 2023) to autonomous agents that navigate codebases and resolve real-world software issues (Jimenez et al., 2024). The clear trajectory is toward end-to-end software development at the repository level, where an agent is asked not for a single edit but for a complete, working system.

Creating a complete repository demands capabilities beyond the localized reasoning and editing exercised by short-horizon tasks such as function completion and issue resolution. Whole-repository development is instead a long-horizon task requiring system-level planning and sustained execution: the model must derive a coherent design from a high-level specification, decompose requirements into interdependent components, establish consistent interfaces, and implement, integrate, and debug the system as a whole. The challenge lies not only in implementing each component correctly, but also in making them work together while satisfying many requirements simultaneously.

To measure this capability, we introduce End2End-SWE (E2E-SWE), a multilingual benchmark for end-to-end repository generation. Given only a natural-language specification and an empty workspace, the agent must produce a complete, installable project whose functional correctness is evaluated by executable hidden tests. Each task is grounded in a real GitHub repository, but its specification and test suite are created from scratch by a software engineer with LLM assistance. Together, they establish a compact behavioral contract for important user-facing functionality while leaving the internal architecture for the agent to design. This setup mirrors a realistic two-stage workflow: a human first collaborates with an LLM to turn project goals into a concrete specification,

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

after which a coding agent autonomously implements that specification as a working repository. E2E-SWE isolates and evaluates the second stage.

Whole-repository generation from a specification alone has been regarded as extremely challenging, with prior benchmarks (Ding et al., 2025; Chen et al., 2026) reporting near-zero pass@1 (the fraction of tasks fully resolved in one attempt). Yet low pass@1 meaningfully reflects model capability only when the underlying tasks are solvable. By solvable, we mean that every behavior enforced by the hidden tests can be derived from the specification, and that any implementation satisfying the stated user-facing contract can pass regardless of how its internals are arranged. Without this property, a failure may reflect an underspecified requirement or an overly rigid test rather than a limitation of the model. Evidence from better-studied coding benchmarks shows this concern is substantial. In a targeted audit of 138 SWE-bench Verified (Jimenez et al., 2024) tasks with recurring model failures, OpenAI found that 59.4% contained material issues in test design or problem descriptions that could reject functionally correct submissions (OpenAI, 2026c). DeepSWE separately reports that, across 789 audited SWE-Bench Pro (Deng et al., 2025) rollouts, the benchmark's hidden tests rejected patches judged to be reasonable solutions in 24.0% of cases (Huang et al., 2026). These audits concern tasks that each target a single issue; a whole-repository task bundles many behavioral requirements, creating many more opportunities for a specification–test mismatch, and a single such mismatch can prevent full resolution. Solvability is therefore especially consequential for full-repository generation.

To uphold this standard, every E2E-SWE task passes through a rigorous post-hoc review-and-repair loop. The loop begins with static audits of the specification and tests, verifying their one-to-one correspondence. Rollout audits then examine failures from real model attempts, distinguishing genuine implementation errors from specification gaps or overly strict tests that static inspection missed.

The final E2E-SWE dataset comprises 186 tasks across 11 programming languages, drawn from reference repositories with a median of 7.4K lines of code. Despite the scale and complexity of these tasks, pass@1 ranges from 11.7% to 67.7% across 13 frontier models. The high-end score shows that meaningful full-resolution rates are attainable when solvability is treated as a first-class design goal, while the broad spread demonstrates that pass@1 effectively distinguishes models by their ability to build complete projects from scratch.

2 The E2E-SWE Benchmark

We now present E2E-SWE in detail. Figure 1 provides an overview of the task lifecycle, spanning task creation, refinement, and evaluation.

2.1 Task Formulation

E2E-SWE adopts the Harbor task format (Harbor Framework Team, 2026). Each task directory holds an `instruction.md` containing the natural-language specification shown to the agent, a `tests/` directory holding the hidden grading suite, a `solution/` directory referencing the source GitHub repository, and a `task.toml` declaring the container image and configuration.

Given an E2E-SWE task, the agent starts from an empty workspace with only the specification document, and must produce a complete repository from scratch with a `setup.sh` script that installs the repository. For grading, we copy the agent's implemented codebase into a fresh new container, run `setup.sh` to install the project, and then execute the hidden test suite. This mirrors how a user clones and installs a repository, to ensure reproducibility. A repository passes only when the code and `setup.sh` the agent committed reproduce the required behavior on their own, not because of transient development-time state such as build cache, scratch files, or incidental environment setup.

Environment and networking. Each task has a dedicated container image with all required dependencies pre-installed. Network access is disabled during both the agent's implementation phase and grading, preventing the agent from retrieving the target repository or related information from the internet. The self-contained image allows both phases to run without any network-dependent setup.

Solution. Each task ships a reference solution at `solution/solve.sh`. It typically clones the original repository, checks out a pinned commit for reproducibility, and then composes the same short `setup.sh` the agent is expected to write.

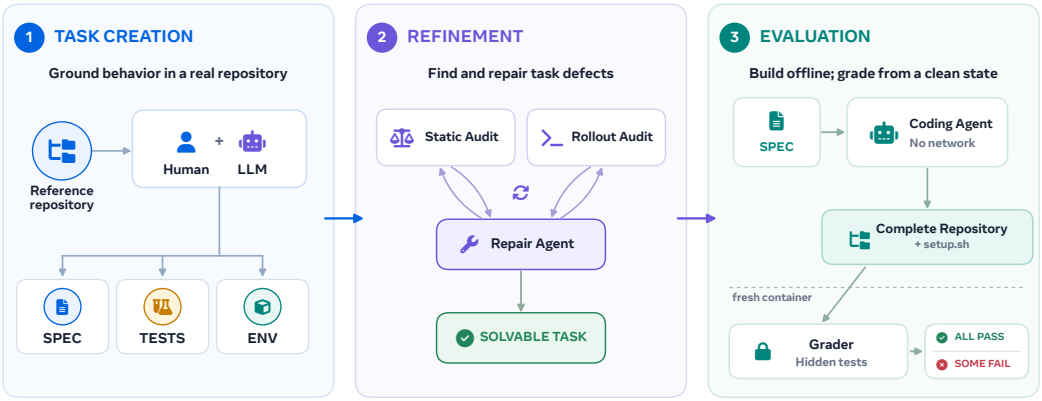


Figure 1: The E2E-SWE task lifecycle. A task author collaborates with an LLM to create a specification, hidden functional test suite, and dependency image from a reference GitHub repository. The task then passes through iterative static and rollout-guided audits, with a repair agent addressing the identified defects. During evaluation, a coding agent receives only the specification and builds a repository in an offline empty workspace. The generated repository is then installed in a fresh grading container and run against the hidden tests; full resolution requires every test to pass.

Grading interface. The test entrypoint `tests/test.sh` must emit a `ctrf.json` report that records the outcome of every individual test. CTRF is a widely supported, language-agnostic test-reporting format, which lets E2E-SWE work with testing frameworks beyond `pytest` and, in principle, generalize to any programming language.

2.2 Task Creation

Each E2E-SWE task is authored by a software engineer with LLM assistance. The agent handles most of the heavy lifting, such as generating test suites that run to hundreds or thousands of lines and drafting the specification. The author plays a complementary role, reviewing the generated content, identifying gaps, steering quality improvements, and making the judgment calls on design choices. Below we describe the step-by-step workflow that authors follow.

2.2.1 Building the Test Suite

Prior work on whole-repository generation often evaluates generated code using the reference repositories' test suites (Zhao et al., 2025; Ding et al., 2025; Zhao et al., 2026). Those tests were designed to validate and maintain a particular codebase, not to assess the functional correctness of an alternative implementation. They may therefore depend on private helpers or other structural details of the reference codebase that a correct reimplementation need not preserve. We instead author a new suite for each task, consulting the original tests only to identify intended behaviors and edge cases.

Each test exercises a realistic user-facing scenario through public interfaces and asserts only observable behavior. This specifies what the repository must do while leaving agents free to choose their internal decomposition, data structures, and algorithms. Authors iteratively measure test coverage on the reference implementation to identify missing functionality, but do not seek to maximize coverage indiscriminately; functionality that cannot be tested or whose inclusion would add only trivial, shallow requirements may be excluded with explicit justification.

2.2.2 Writing the Specification

Authors write the task specification only after creating the hidden test suite. The specification is test-driven rather than an exhaustive description of the reference implementation. It provides sufficient information for an agent to implement every behavior required by the tests. Implementation details not pinned by the tests, such as internal decomposition, private helpers, and source-tree layout, are deliberately left unspecified.

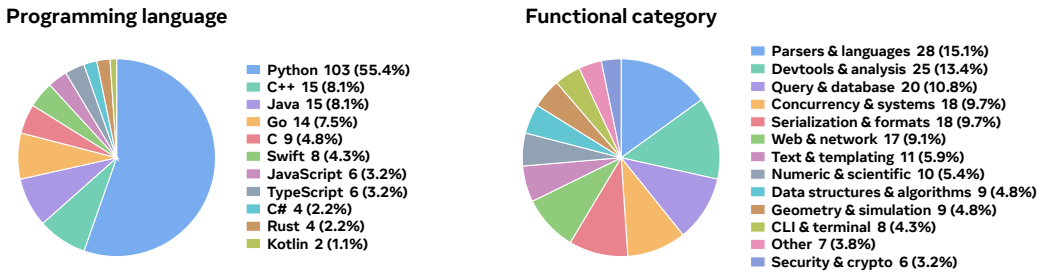


Figure 2: Composition of the E2E-SWE corpus by programming language and functional category.

The correspondence between specification and tests is central to task solvability. Every tested behavior must be stated in the specification or follow from a standard domain convention, and every documented behavior must be exercised by the tests. Authors audit this alignment assertion by assertion; when a test depends on a choice left open by the specification, we either make that choice an explicit requirement or relax the test to accept any reasonable choice.

2.3 Quality Refinement

After authoring, each task enters an automated refinement loop spanning multiple rounds. In each round, specialized LLM reviewers inspect the task from complementary perspectives, and a revision agent updates the specification or tests to address their findings.

Static Review. Prior to model evaluation, we conduct two types of static review with independent review agents. *Test Quality* assesses the hidden suite against the reference repository without reading the specification, checking that it covers major user-facing functionality with substantive, nonredundant, and implementation-agnostic assertions. *Spec Fairness* audits the correspondence between specification and tests without access to the reference repository, verifying that each asserted behavior is explicitly documented or follows a standard domain convention and flagging tests that depend on choices the specification leaves open.

Rollout-Guided Review. After static review, we run coding agents on each task and collect the test failures from their rollouts. A reviewer examines each failure to determine whether the generated implementation fails to meet the stated requirements or satisfies them but is nevertheless rejected by the test. The former is a genuine model error; the latter indicates a task defect such as an underspecified contract, an overly rigid assertion, or a flaky test. By observing how independently generated code behaves under the tests, rollout-guided review can expose defects that are difficult to identify through static inspection alone.

2.4 Dataset Statistics

Our final dataset consists of 186 tasks spanning 11 programming languages and 13 functional categories. As Figure 2 shows, E2E-SWE is diverse both in programming language and in the functionality that agents must build, ranging from parsers, developer tools, and database systems to networking, scientific computing, simulation, and cryptography.

The complexity of a E2E-SWE task stems from both the breadth of behavior expressed in its specification and the substantial implementation it requires the agent to reconstruct. Table 1 characterizes these complementary dimensions through the scope of the authored task artifacts and the size and maturity of their reference repositories.

3 Experiments

3.1 Experimental Setup

Models. We evaluate thirteen frontier models across six model families: Claude, GPT, Muse Spark, Gemini, Kimi, and GLM. We select the strongest reasoning-effort setting that runs reliably within

Table 1: Statistics for E2E-SWE tasks and their reference repositories, reported at the 10th, 50th, and 90th percentiles.

Benchmark Task	P10	P50	P90
Hidden tests	21	38	96
Spec (chars)	8,460	16,882	29,206
Reference Repository			
Code files	12	46	144
Code lines	2,398	7,360	25,457
GitHub stars	130	776	5,211
Commits	186	754	2,512

Table 2: Evaluation results on the 186-task E2E-SWE corpus, aggregated over 4 independent runs. Pass@1 is our main performance metric, measuring the percentage of tasks fully resolved. †Claude Fable 5 uses `high` because `xhigh` often exceeds the 128K output-token limit per request.

Model	Effort	Pass@1	Agent turns	Output tokens (k)
Claude Fable 5† (Anthropic, 2026a)	high	<u>65.3</u>	56.3	175.1
Claude Opus 5 (Anthropic, 2026d)	xhigh	**67.7**	120.2	**263.2**
Claude Sonnet 5 (Anthropic, 2026e)	xhigh	43.1	<u>176.6</u>	<u>231.0</u>
Claude Opus 4.8 (Anthropic, 2026c)	xhigh	51.9	87.6	208.5
Claude Opus 4.6 (Anthropic, 2026b)	high	22.6	**219.6**	120.4
GPT-5.6 Sol (OpenAI, 2026b)	max	61.2	67.7	92.6
GPT-5.6 Terra (OpenAI, 2026b)	max	51.8	71.9	116.7
GPT-5.5 (OpenAI, 2026a)	xhigh	46.1	59.5	117.9
Muse Spark 1.3 (Meta, 2026)	max	45.2	98.1	157.0
Gemini 3.8 Flash (Google, 2026)	high	36.7	141.2	165.8
Gemini 3.6 Flash (Google DeepMind, 2026)	high	11.7	85.1	114.5
Kimi K3 (Kimi Team, 2026)	high	42.3	94.9	170.6
GLM-5.2 (Z.ai, 2026)	high	22.2	137.7	191.4

the 128K per-request output-token limit enforced by API providers, since some models at `max` effort may exceed this limit due to excessively long reasoning in a single turn, causing API call failures. We run each model four times independently over the full corpus and report the average.

Agent scaffold. All models are evaluated with a simple ReAct agent, similar to mini-SWE-agent (Yang et al., 2024) except for two additional tools: an edit tool for efficiently viewing and modifying long files and a submit tool for explicitly signaling task completion. Each rollout has a 4-hour wall-clock budget, and is capped at 1 million context tokens and 1,000 agent turns without context compaction. Neither the token nor agent-turn limit was reached in practice.

Metric. Our primary metric is pass@1: the fraction of tasks fully resolved by a single rollout, where resolution requires the generated repository to pass every hidden test. We do not award partial credit for passing only a subset of a task's tests. A repository is an integrated software artifact, and one missing critical behavior can leave it unusable even when many other tests pass. Moreover, test suites differ in their number and granularity, so fractional test scores can reflect how behavior was decomposed into assertions rather than how much of the repository was correctly reconstructed. Full resolution instead weights each task equally and directly measures whether the model produced a complete implementation of the specified contract.

3.2 Main Results

Table 2 reports full-repository resolution together with the average trajectory length and token usage of each model. Claude Opus 5 achieves the highest pass@1, followed closely by Claude Fable 5, with GPT-5.6 Sol ranking third. Even the strongest model leaves nearly one third of the tasks unresolved, indicating substantial headroom in whole-repository generation. Meanwhile, performance is

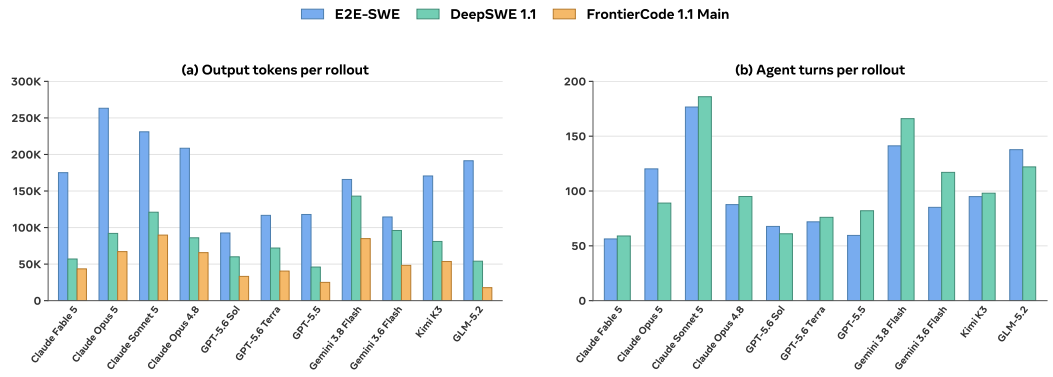


Figure 3: Rollout size across models using matched reasoning-effort settings on E2E-SWE, DeepSWE, and FrontierCode. Numbers for DeepSWE and FrontierCode are taken from their official leaderboards; FrontierCode does not report agent-turn counts.

broadly distributed rather than concentrated near either a floor or a ceiling, giving E2E-SWE strong distinguishing power among current frontier models.

Models differ widely in how effectively they convert interaction and generation budgets into working software. Opus 4.6 resolves far fewer tasks than GPT-5.6 Sol despite using more than three times as many turns. The contrast extends to output-token usage: Fable 5 nearly matches Opus 5 while using roughly two thirds as many tokens, whereas Sonnet 5 uses almost as many tokens as Opus 5 but resolves substantially fewer tasks.

To characterize the complexity of full-repository generation, Figure 3 compares rollout size on E2E-SWE with two advanced software-engineering benchmarks. For every matched model, E2E-SWE elicits more output tokens than DeepSWE (Huang et al., 2026) and FrontierCode (Lu et al., 2026), consistent with generating a complete repository rather than implementing a single-PR change. Turn counts on E2E-SWE are nevertheless similar to those on DeepSWE, largely because E2E-SWE removes the initial codebase-exploration stage: it provides the complete specification upfront, whereas DeepSWE agents must navigate an existing codebase to gather context and locate the issue.

Impact of reasoning effort. We additionally sweep reasoning-effort levels for Claude Opus 4.8 and GPT-5.5, as shown in Figure 4. From `low` through `xhigh`, both models use progressively more output tokens and consistently achieve higher pass@1, indicating that whole-repository generation benefits strongly from longer reasoning. The gains eventually saturate: moving Opus 4.8 from `xhigh` to `max` consumes considerably more tokens without further improving pass@1.[1]

4 Analysis

4.1 Task Solvability

To separate signal from noise in our evaluation results, we assess whether test failures reflect genuine model limitations or flaws in the task itself. We conduct the same rollout audit described in Section 2.3 over one full-corpus run from each of four models: Claude Opus 5, Opus 4.6, GPT-5.6 Sol, and Kimi K3. For each failure, an independent LLM judge (Claude Opus 5 at `max` effort) examines the specification, test, traceback, and generated implementation, then attributes the failure either to a model implementation error or to a task defect such as an underspecified requirement or over-rigid test.

Across the $186 \times 4 = 744$ model attempts, 29 (3.9%) contain at least one failure that the judge attributes to the task. Excluding these failures from grading would flip 16 outcomes (2.2% of all

[1]Only four tasks reached the 128K output-token limit for Opus 4.8 at `max`, too few to change the overall saturation pattern.

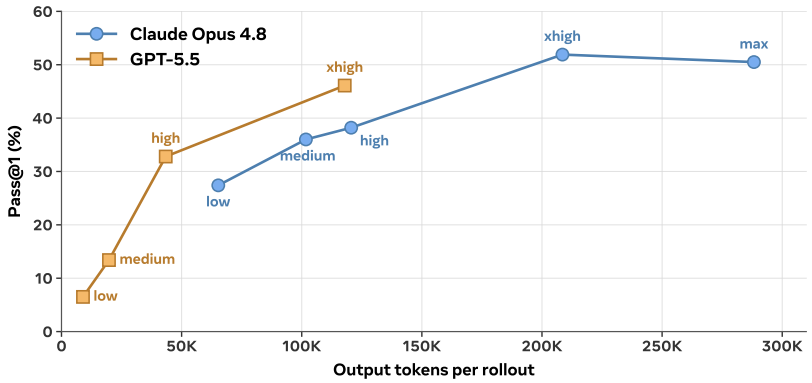
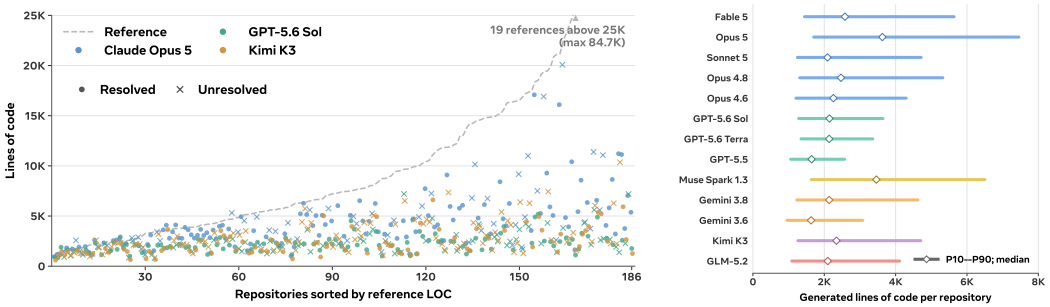


Figure 4: Inference-time scaling with reasoning effort. Each point shows pass@1 and average output tokens at one effort setting; annotations denote effort levels and colors distinguish models.



(a) Per-task size against the reference.

(b) Size distributions across models.

Figure 5: Generated codebase sizes measured in nonblank implementation lines. *Left*: repositories are ordered by reference size; colors identify models, circles denote resolved tasks, crosses denote unresolved tasks, and the reference curve is capped at 25K LOC. *Right*: lines span the 10th to 90th percentiles and diamonds mark medians.

attempts) from unresolved to resolved; the other 13 are unaffected because they also involve failures attributed to the model.

Notice that these rates should be interpreted as judge attributions rather than definitive defect counts. E2E-SWE tasks contain many interacting requirements, and the judge may overlook that a tested behavior follows from requirements distributed across the specification, thereby misclassifying a genuine model error as a task defect.

4.2 Generated Codebases

We next examine the size of the generated codebases. Figure 5a shows that models generally produce substantially fewer lines of code than the reference repositories, particularly as reference size grows. Within each model, resolved and unresolved outcomes are interspersed across the generated-size range, revealing no strong relationship between codebase size and success.

Across all models, Figure 5b reveals a clear family-specific pattern. Although the Claude and GPT distributions overlap near their medians, Claude models more often produce codebases at the larger end of the range, consistent with their higher output-token usage in Table 2. Generated codebase size is also largely orthogonal to model capability. Opus 4.6 and Sonnet 5, for example, produce similarly sized codebases despite their large pass@1 gap. Together, these observations suggest that generated LOC reflects a model's coding style more than its correctness.

Table 3: Reasoning distribution and 12-gram containment. **(a)** Reasoning is measured as output tokens excluding tool-calls. For each trajectory, Total (k) and Peak (k) are the sum and max of reasoning tokens across agent turns, respectively. Peak share is peak length divided by total reasoning, and peak position is the 1-based index of the peak turn. We report means for total reasoning and peak share, median [P10–P90] for peak length, and median for peak position. **(b)** 12-gram containment of model-generated code against the reference repository ($n = 186$ tasks, one rollout per model). Max is the highest task-level containment observed for each model, and the last two columns count tasks above the indicated threshold.

Model	(a) Reasoning distribution				(b) Code containment		
	Total (k)	Peak (k) P50 [P10–P90]	Peak Share	Peak Pos.	Max	$> 50\%$	$> 75\%$
Claude Fable 5	123.5	63.0 [23.9–104.2]	52.2%	2	98.38%	30	13
Claude Opus 5	180.1	44.0 [24.1–82.4]	28.1%	4	97.77%	25	10
Claude Sonnet 5	170.3	28.2 [15.1–53.1]	20.5%	12	91.87%	2	1
Claude Opus 4.8	158.2	34.5 [14.8–65.3]	26.1%	8	98.49%	24	8
Claude Opus 4.6	37.5	2.1 [0.8–7.6]	8.3%	68	33.34%	0	0
GPT-5.6 Sol	47.9	7.3 [4.4–12.5]	17.2%	3	14.35%	0	0
GPT-5.6 Terra	68.5	9.9 [5.4–14.8]	15.1%	4	22.60%	0	0
GPT-5.5	80.4	12.6 [9.5–16.5]	18.5%	6	22.78%	0	0
Muse Spark 1.3	86.7	32.7 [14.7–65.7]	43.5%	4	11.09%	0	0
Gemini 3.8 Flash	104.8	21.7 [10.3–44.3]	24.4%	18	30.50%	0	0
Gemini 3.6 Flash	60.2	18.2 [7.6–32.5]	34.2%	6	31.51%	0	0
Kimi K3	109.9	54.2 [22.7–95.2]	55.9%	3	94.52%	4	2
GLM-5.2	138.5	26.0 [8.9–64.9]	26.5%	12	15.83%	0	0

4.3 Reasoning Distribution

Table 3(a) characterizes how reasoning is distributed across agent turns, reporting total reasoning together with the length, share, and position of each trajectory's longest reasoning turn. The dominant pattern is a single, front-loaded reasoning peak. For 11 of the 13 models, the longest reasoning turn typically occurs within the first 12 agent turns; for six of them, it occurs within the first four, even though trajectories span dozens or hundreds of turns. These peaks can dominate a trajectory's reasoning volume: the largest reported P90 exceeds 100K tokens, and peak share exceeds 50% in the most concentrated cases.

We attribute this shared temporal structure to the E2E-SWE task formulation. Agents receive the complete specification upfront and have no existing codebase to explore, encouraging them to form a repository-wide plan early and spend subsequent turns implementing, testing, and revising it.

4.4 Memorization

Each E2E-SWE task is grounded in a public GitHub repository whose source code may have appeared in the models' pretraining data. This raises a natural question: how often does an agent solve a task by recalling the exact reference code from its memory rather than building from the given specification? To quantify this, we propose a metric of *12-gram containment*. Within each code file, we extract overlapping 12-token windows with a stride of 1, using a model-agnostic, code-aware tokenizer. Let G_{12} and R_{12} denote the resulting multisets for the generated and reference code, respectively. Containment is defined as $|G_{12} \cap R_{12}|/|G_{12}|$. Table 3(b) reports the maximum observed containment and the number of tasks exceeding 50% and 75% containment for each model.

The strongest memorization signal is observed among Claude Fable 5, Opus 5, and Opus 4.8. These models exceed 50% containment on 24–30 tasks and 75% on 8–13 tasks; by contrast, eight other models have no task above 50% containment. We further compare the grading outcomes within the $> 50\%$ task group against the rest for those three models. On the absolute scale, the resolve-rate deltas are -0.6 pp, $+9.6$ pp, and $+5.3$ pp for Fable 5, Opus 5, and Opus 4.8, respectively. Contrary to intuition, substantial reference-code overlap is far from guaranteeing resolution of the task.

5 Related Work

Repository-level coding was initially studied in settings where the project is largely implemented. RepoBench (Liu et al., 2024), RepoCoder (Zhang et al., 2023), CrossCodeEval (Ding et al., 2023), and DevEval (Li et al., 2024) evaluate line- or function-level completion with cross-file context. SWE-bench (Jimenez et al., 2024) and its variants (Yang et al., 2025; Zan et al., 2025; Deng et al., 2025; Huang et al., 2026; Lu et al., 2026), instead target PR-level code changes to an existing repository.

Recent benchmarks push beyond localized edits toward whole-repository construction across varied task formulations and software domains. DevBench (Li et al., 2025) independently evaluates four stages of repository development—design, environment setup, implementation, and testing—supplying each with golden inputs rather than artifacts produced in preceding stages. Commit0 (Zhao et al., 2025) constructs tasks from 54 Python libraries by removing function and class bodies, leaving signatures and docstrings as context for reconstructing the implementations. RepoZero (Zhang et al., 2026) derives 600 task instances from 35 source repositories, with each instance associated with one API-usage scenario. Domain-specific benchmarks cover microservices in RepoGenesis (Peng et al., 2026), interactive applications in ProjectEval (Liu et al., 2025) and E2EDev (Liu et al., 2026), command-line tools in CLI-Tool-Bench (Hu et al., 2026), and research codebases in Paper2Code (Seo et al., 2025) and PaperBench (Starace et al., 2025). None of these benchmarks provides a domain-general evaluation of end-to-end repository creation from a project specification alone.

NL2Repo-Bench (Ding et al., 2025) is the closest prior benchmark to E2E-SWE in both task formulation and scale. The benchmark contains 104 tasks, each asking an agent to generate a complete Python repository from scratch given only a specification document. The resulting workspace is then graded by the original repository's test suite. In our evaluation, NL2Repo-Bench's pass@1 clusters near 10% for all models except Claude Fable 5 and Opus 5, providing limited separation among frontier models (Appendix A). A possible explanation is that underspecified requirements and overly strict tests make many tasks effectively unsolvable. By contrast, E2E-SWE pairs independently authored tests with test-driven specifications and iteratively reviews their behavioral alignment; empirically, it clearly separates the 13 frontier models evaluated.

ProgramBench (Yang et al., 2026) and MirrorCode (Adamczewski et al., 2026) study a complementary formulation that supplements written documentation with black-box access to a reference executable. Their tasks jointly evaluate behavioral requirement discovery through probing and full-codebase implementation. In some cases, determining the full set of required behaviors can be more challenging than implementing them once known, making probing the bottleneck for benchmark performance. In contrast, E2E-SWE states the tested behaviors upfront, disentangling repository implementation from requirement discovery. ProgramBench and MirrorCode also require the reference repository's relevant functionality to be exposed through a standalone, queryable program, limiting their applicability to repositories organized primarily around reusable APIs or other non-executable interfaces.

6 Conclusion

We introduced E2E-SWE, a benchmark of 186 tasks across 11 programming languages for generating complete repositories from natural-language specifications. Its specifications and executable hidden tests are created from scratch and iteratively reviewed and repaired, making strict full-repository resolution meaningful while preserving freedom in internal design. Across 13 models, pass@1 ranges from 11.7% to 67.7%, yielding clear model separation and ample headroom; a post-hoc LLM audit attributes outcome-changing failures to task issues in only 2.2% of attempts. Together, these results show that E2E-SWE offers a demanding yet measurable test of long-horizon software engineering capability in whole-repository generation.

AI use statement

We used generative AI tools to construct and refine benchmark tasks (Sections 2.2 and 2.3), clean and reformat benchmark data, and support qualitative analysis, including the post-hoc solvability study

(Section 4.1). Additionally, generative AI assisted with writing and debugging code, creating and modifying figures, identifying and summarizing relevant literature, and preparing the manuscript. Generative AI was not used for research ideation; the benchmark concept and research goals originated with the authors. We have reviewed all AI-assisted work, verified the resulting artifacts, analyses, citations, and claims, and made all final methodological and editorial decisions.

REFERENCES

Tom Adamczewski, David Owen, David Rein, Florian Brand, Giles Edkins, Allen Hart, and Daniel O'Connell. Mirrorcode: AI can rebuild entire programs from behavior alone. *CoRR*, abs/2606.30182, 2026. URL https://arxiv.org/abs/2606.30182.

Anthropic. Claude Fable 5 and Claude Mythos 5 system card, 2026a. URL https://www.anthropic.com/claude-fable-5-system-card.

Anthropic. Claude Opus 4.6, 2026b. URL https://www.anthropic.com/news/claude-opus-4-6.

Anthropic. Introducing Claude Opus 4.8, 2026c. URL https://www.anthropic.com/news/claude-opus-4-8.

Anthropic. Introducing Claude Opus 5, 2026d. URL https://www.anthropic.com/news/claude-opus-5.

Anthropic. Claude Sonnet 5 system card, 2026e. URL https://www.anthropic.com/document/claude-sonnet-5-system-card.

Guoxin Chen, Fanzhe Meng, Jiale Zhao, Minghao Li, Daixuan Cheng, Huatong Song, Jie Chen, Yuzhi Lin, Hui Chen, Xin Zhao, Ruihua Song, Chang Liu, Cheng Chen, Kai Jia, and Ji-Rong Wen. Beyondswe: Can current code agent survive beyond single-repo bug fixing? *CoRR*, abs/2603.03194, 2026. doi: 10.48550/ARXIV.2603.03194. URL https://doi.org/10.48550/arXiv.2603.03194.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021. URL https://arxiv.org/abs/2107.03374.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. Swe-bench pro: Can AI agents solve long-horizon software engineering tasks? *CoRR*, abs/2509.16941, 2025. doi: 10.48550/ARXIV.2509.16941. URL https://doi.org/10.48550/arXiv.2509.16941.

Jingzhe Ding, Shengda Long, Changxin Pu, Huan Zhou, Hongwan Gao, Xiang Gao, Chao He, Yue Hou, Fei Hu, Zhaojian Li, Weiran Shi, Zaiyuan Wang, Daoguang Zan, Chenchen Zhang, Xiaoxu Zhang, Qizhi Chen, Xianfu Cheng, Bo Deng, Qingshui Gu, Kai Hua, Juntao Lin, Pai Liu, Mingchen Li, Xuanguang Pan, Zifan Peng, Yujia Qin, Yong Shan, Zhewen Tan, Weihao Xie, Zihan Wang, Yishuo Yuan, Jiayu Zhang, Enduo Zhao, Yunfei Zhao, He Zhu, Chenyang Zou, Ming Ding, Jianpeng Jiao, Jiaheng Liu, Minghao Liu, Qian Liu, Chongyao Tao, Jian Yang, Tong Yang, Zhaoxiang Zhang, Xinjie Chen, Wenhao Huang, and Ge Zhang. Nl2repo-bench: Towards long-horizon repository generation evaluation of coding agents. *CoRR*, abs/2512.12730, 2025. doi: 10.48550/ARXIV.2512.12730. URL https://doi.org/10.48550/arXiv.2512.12730.

Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine

(eds.), *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/920f2dced7d32ab2ba2f1970bc306af6-Abstract-Datasets_and_Benchmarks.html.

Google. Introducing Gemini 3.8 Flash and Gemini 3.8 Flash Cyber, 2026. URL https://blog.google/innovation-and-ai/models-and-research/gemini-models/3-8-flash-and-3-8-flash-cyber/.

Google DeepMind. Gemini 3.6 Flash model card, 2026. URL https://deepmind.google/models/model-cards/gemini-3-6-flash/.

Harbor Framework Team. Harbor: A framework for evaluating and optimizing agents and models in container environments, 2026. URL https://doi.org/10.5281/zenodo.20953922.

Ruida Hu, Xinchen Wang, Chao Peng, Cuiyun Gao, and David Lo. Evaluating llm-based 0-to-1 software generation in end-to-end CLI tool scenarios. *CoRR*, abs/2604.06742, 2026. URL https://arxiv.org/abs/2604.06742.

Wenqi Huang, Charley Lee, Leonard Tng, and Serena Ge. DeepSWE: Measuring frontier coding agents on original, long-horizon engineering tasks. *CoRR*, abs/2607.07946, 2026. doi: 10.48550/ARXIV.2607.07946. URL https://doi.org/10.48550/arXiv.2607.07946.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Kimi Team. Kimi K3: Open frontier intelligence. *CoRR*, abs/2607.24653, 2026. URL https://arxiv.org/abs/2607.24653.

Bowen Li, Wenhan Wu, Ziwei Tang, Lin Shi, John Yang, Jinyang Li, Shunyu Yao, Chen Qian, Binyuan Hui, Qicheng Zhang, Zhiyin Yu, He Du, Ping Yang, Dahua Lin, Chao Peng, and Kai Chen. Prompting large language models to tackle the full software development lifecycle: A case study. In Owen Rambow, Leo Wanner, Marianna Apidianaki, Hend Al-Khalifa, Barbara Di Eugenio, and Steven Schockaert (eds.), *Proceedings of the 31st International Conference on Computational Linguistics, COLING 2025, Abu Dhabi, UAE, January 19-24, 2025*, pp. 7511–7531. Association for Computational Linguistics, 2025. URL https://aclanthology.org/2025.coling-main.502/.

Jia Li, Ge Li, Yunfei Zhao, Yongmin Li, Huanyu Liu, Hao Zhu, Lecheng Wang, Kaibo Liu, Zheng Fang, Lanshen Wang, Jiazheng Ding, Xuanming Zhang, Yuqi Zhu, Yihong Dong, Zhi Jin, Binhua Li, Fei Huang, Yongbin Li, Bin Gu, and Mengfei Yang. Deveval: A manually-annotated code generation benchmark aligned with real-world code repositories. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, volume ACL 2024 of *Findings of ACL*, pp. 3603–3614. Association for Computational Linguistics, 2024. doi: 10.18653/V1/2024.FINDINGS-ACL.214. URL https://doi.org/10.18653/v1/2024.findings-acl.214.

Jingyao Liu, Chen Huang, Zhizhao Guan, Wenqiang Lei, and Yang Deng. E2edev: Benchmarking large language models in end-to-end software development task. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2026, San Diego, California, United States, July 2-7, 2026*, pp. 35032–35068. Association for Computational Linguistics, 2026. URL https://aclanthology.org/2026.acl-long.1618/.

Kaiyuan Liu, Youcheng Pan, Yang Xiang, Daojing He, Jing Li, Yexing Du, and Tianrun Gao. Projecteval: A benchmark for programming agents automated evaluation on project-level code generation. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar

(eds.), *Findings of the Association for Computational Linguistics, ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, volume ACL 2025 of *Findings of ACL*, pp. 20205–20221. Association for Computational Linguistics, 2025. doi: 10.18653/V1/2025.FINDINGS-ACL.1036. URL https://doi.org/10.18653/v1/2025.findings-acl.1036.

Tianyang Liu, Canwen Xu, and Julian J. McAuley. Repobench: Benchmarking repository-level code auto-completion systems. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=pPjZIOuQuF.

Eric Lu, Ben Pan, Deniz Birlikci, Sam Lee, Ray Wang, Rohan Choudhury, Fermi Ma, TC Qin, Carlo Baronio, and Silas Alberti. Introducing FrontierCode. Cognition blog, 2026. URL https://cognition.com/blog/frontier-code. Accessed 2026-08-18.

Meta. Introducing Muse Spark 1.3, 2026. URL https://research.meta.ai/blog/introducing-muse-spark-1-3.

Vijayaraghavan Murali, Chandra Maddila, Imad Ahmad, Michael Bolin, Daniel Cheng, Negar Ghorbani, Renuka Fernandez, Nachiappan Nagappan, and Peter C. Rigby. AI-assisted code authoring at scale: Fine-tuning, deploying, and mixed methods evaluation. *CoRR*, abs/2305.12050, 2023. doi: 10.48550/ARXIV.2305.12050. URL https://doi.org/10.48550/arXiv.2305.12050.

OpenAI. GPT-5.5 system card, 2026a. URL https://openai.com/index/gpt-5-5-system-card/.

OpenAI. GPT-5.6: Frontier intelligence that scales with your ambition, 2026b. URL https://openai.com/index/gpt-5-6/.

OpenAI. Why SWE-bench Verified no longer measures frontier coding capabilities. https://openai.com/index/why-we-no-longer-evaluate-swe-bench-verified/, 2026c. OpenAI blog. Accessed 2026-08-27.

Zhiyuan Peng, Xin Yin, Pu Zhao, Fangkai Yang, Lu Wang, Ran Jia, Xu Chen, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. Repogenesis: Benchmarking end-to-end microservice generation from readme to repository. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2026, San Diego, California, United States, July 2-7, 2026*, pp. 9161–9189. Association for Computational Linguistics, 2026. URL https://aclanthology.org/2026.acl-long.415/.

Minju Seo, Jinheon Baek, Seongyun Lee, and Sung Ju Hwang. Paper2code: Automating code generation from scientific papers in machine learning. *CoRR*, abs/2504.17192, 2025. doi: 10.48550/ARXIV.2504.17192. URL https://doi.org/10.48550/arXiv.2504.17192.

Giulio Starace, Oliver Jaffe, Dane Sherburn, James Aung, Jun Shern Chan, Leon Maksin, Rachel Dias, Evan Mays, Benjamin Kinsella, Wyatt Thompson, Johannes Heidecke, Amelia Glaese, and Tejal Patwardhan. Paperbench: Evaluating ai's ability to replicate AI research. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025*, volume 267 of *Proceedings of Machine Learning Research*. PMLR / OpenReview.net, 2025. URL https://proceedings.mlr.press/v267/starace25a.html.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. In Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (eds.), *Advances in Neural Information Processing Systems 37: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html.

John Yang, Carlos E. Jimenez, Alex L. Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R. Narasimhan, Diyi Yang, Sida Wang, and Ofir Press. Swe-bench multimodal: Do AI systems generalize to visual software domains? In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. URL https://openreview.net/forum?id=riTiq3i21b.

John Yang, Kilian Lieret, Jeffrey Ma, Parth Thakkar, Dmitrii Pedchenko, Sten Sootla, Emily McMilin, Pengcheng Yin, Rui Hou, Gabriel Synnaeve, Diyi Yang, and Ofir Press. Program-bench: Can language models rebuild programs from scratch? *CoRR*, abs/2605.03546, 2026. doi: 10.48550/ARXIV.2605.03546. URL https://doi.org/10.48550/arXiv.2605.03546.

Z.ai. GLM-5.2, 2026. URL https://docs.z.ai/guides/llm/glm-5.2.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, Lu Chen, Qi Liu, Xiaojian Zhong, Aoyan Li, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, Rui Long, Kai Shen, and Liang Xiang. Multi-swe-bench: A multilingual benchmark for issue resolving. *CoRR*, abs/2504.02605, 2025. doi: 10.48550/ARXIV.2504.02605. URL https://doi.org/10.48550/arXiv.2504.02605.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval and generation. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, pp. 2471–2484. Association for Computational Linguistics, 2023. doi: 10.18653/V1/2023.EMNLP-MAIN.151. URL https://doi.org/10.18653/v1/2023.emnlp-main.151.

Zhaoxi Zhang, Yiming Xu, Jiahui Liang, Weikang Li, Xiaoshuai Chen, Liwei Qian, Xin Pei, Jizhou Huang, Run Sun, and Yunfang Wu. Repozero: Can llms generate a code repository from scratch? *CoRR*, abs/2605.07122, 2026. doi: 10.48550/ARXIV.2605.07122. URL https://doi.org/10.48550/arXiv.2605.07122.

Jiale Zhao, Guoxin Chen, Fanzhe Meng, Wayne Xin Zhao, Ruihua Song, Ji-Rong Wen, and Kai Jia. Denovoswe: Scaling long-horizon environments for generating entire repositories from scratch. *CoRR*, abs/2606.10728, 2026. doi: 10.48550/ARXIV.2606.10728. URL https://doi.org/10.48550/arXiv.2606.10728.

Wenting Zhao, Nan Jiang, Celine Lee, Justin T. Chiu, Claire Cardie, Matthias Gallé, and Alexander M. Rush. Commit0: Library generation from scratch. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. URL https://openreview.net/forum?id=MMwaQEVsAg.

A EVALUATION ON NL2REPO-BENCH

We evaluate all 13 models from Section 3.2 on NL2Repo-Bench (Ding et al., 2025) using the same agent scaffold and reasoning-effort settings. To prevent agents from retrieving public reference implementations in the open-Internet setting, we disable network access and start each rollout in the grading image with all required dependencies pre-installed.

Table 4: Evaluation results on the 104-task NL2Repo-Bench corpus. Each model is evaluated in one full-corpus run. Pass@1 is the percentage of tasks fully resolved; agent turns and output tokens are averages per rollout.

Model	Effort	Pass@1	Agent turns	Output tokens (k)
Claude Fable 5 (Anthropic, 2026a)	high	<u>19.2</u>	53.3	165.3
Claude Opus 5 (Anthropic, 2026d)	xhigh	**22.1**	82.1	**184.6**
Claude Sonnet 5 (Anthropic, 2026e)	xhigh	13.5	**167.5**	<u>171.4</u>
Claude Opus 4.8 (Anthropic, 2026c)	xhigh	13.5	83.8	158.7
Claude Opus 4.6 (Anthropic, 2026b)	high	8.7	129.8	68.5
GPT-5.6 Sol (OpenAI, 2026b)	max	7.7	73.7	95.7
GPT-5.6 Terra (OpenAI, 2026b)	max	8.7	61.4	100.6
GPT-5.5 (OpenAI, 2026a)	xhigh	12.5	33.9	74.2
Muse Spark 1.3 (Meta, 2026)	max	9.6	84.7	88.9
Gemini 3.8 Flash (Google, 2026)	high	8.7	<u>166.6</u>	134.4
Gemini 3.6 Flash (Google DeepMind, 2026)	high	8.7	97.6	87.1
Kimi K3 (Kimi Team, 2026)	high	14.4	78.8	135.4
GLM-5.2 (Z.ai, 2026)	high	9.6	123.0	134.4

Table 4 shows a compressed performance range. Except for Claude Opus 5 and Fable 5, every model resolves only 8–15 tasks, with many scores separated by just one or two tasks. Pass@1 is lower than on E2E-SWE for every model. However, two indicators suggest that NL2Repo-Bench is unlikely to be more complex than E2E-SWE. The reported median reference-repository size falls between 1.5K and 4K lines of code, well below E2E-SWE's median of 7,360, and models also generate fewer output tokens per rollout on NL2Repo-Bench. Together, these observations suggest that its low, tightly clustered scores reflect factors beyond task complexity, limiting its resolution among frontier models.