SWE-AGI: Benchmarking Specification-Driven Software Construction with MoonBit in the Era of Autonomous Agents

Zhirui Zhang[1,*], Hongbo Zhang[1,2,*], Haoxiang Fei[1,*], Zhiyuan Bao[1], Yubin Chen[1], Zhengyu Lei[1], Ziyue Liu[1], Yixuan Sun[1], Mingkun Xiao[1], Zihang Ye[1], Yu Zhang[1], Hongcheng Zhu[1], Yuxiang Wen[1], Heung-Yeung Shum[1,2,+]

[1]International Digital Economy Academy
[2]The Hong Kong University of Science and Technology
[1]zrustc11@gmail.com,hzhangcy@connect.ust.hk,feihaoxiang@idea.edu.cn
[*]Equal contribution.
[+]Corresponding author: hshum@ust.hk

Abstract

Although large language models (LLMs) have demonstrated impressive coding capabilities, their ability to autonomously build production-scale software from explicit specifications remains an open question. In this paper, we introduce SWE-AGI, the first open-source benchmark for evaluating end-to-end, specification-driven construction of software systems written in MoonBit. SWE-AGI tasks require LLM-based agents to implement a range of software systems, including parsers, interpreters, binary decoders, and SAT solvers, strictly from authoritative standards and RFCs under a fixed API scaffold. Each task involves implementing 10^3–10^4 lines of core logic, corresponding to weeks or months of engineering effort for an experienced human developer. By leveraging the nascent MoonBit ecosystem, SWE-AGI minimizes data leakage, forcing agents to rely on long-horizon architectural reasoning rather than code retrieval. Across frontier models, gpt-5.3-codex achieves the best overall performance (solving 19/22 tasks, 86.4%), outperforming claude-opus-4.6 (15/22, 68.2%), and kimi-2.5 exhibits the strongest performance among open-source models. However, performance degrades sharply with increasing task difficulty, particularly on hard, specification-intensive systems. Behavioral analysis further reveals that as codebases scale, code reading, rather than writing, becomes the dominant bottleneck in AI-assisted development. Overall, while specification-driven autonomous software engineering is increasingly viable, substantial challenges remain before it can reliably support production-scale development.

1. Introduction

Large language models (LLMs) (Anthropic, 2025; DeepSeek Team et al., 2025; Gemini Team et al., 2025; Kimi Team et al., 2025; OpenAI, 2025; Qwen Team et al., 2025) are increasingly deployed as software engineering (SWE) agents: they read specifications, write and refactor code, run tests, and iterate over long trajectories. As this workflow becomes a practical interface for building and maintaining software, evaluation must move past single-shot code completion to address a more fundamental challenge: can an AI system autonomously carry out a production-scale implementation from explicit requirements to generate a correct, robust, and maintainable

codebase?

Most existing benchmarks only partially capture this end-to-end capability. Function- and problem-level tasks (Austin et al., 2021; Chen et al., 2021) are often short-horizon and can be solved via pattern matching or overfitting to limited tests. Repository-issue benchmarks (Deng et al., 2025; Jimenez et al., 2023; Yang et al., 2024) more closely reflect iterative development, but their results are frequently confounded by repository-specific conventions, hidden degrees of freedom in tooling, and difficult-to-control training-data overlap. To measure autonomy at this level, a benchmark should instead be specification-grounded, production-scale, and evaluated using deterministic, human-validated tests under a standardized interface.

In this paper, we introduce SWE-AGI[1], the first open-source benchmark for assessing autonomous software engineering through specification-driven, from-scratch system construction in MoonBit, a modern programming language with a nascent ecosystem. Leveraging MoonBit's native support for spec-first development and its integrated toolchain (MoonBit Team, 2025), SWE-AGI tasks require LLM-based agents to implement production-grade, standards-compliant systems in MoonBit strictly from authoritative specifications within a fixed API scaffold. Concretely, MoonBit supports declaration-first workflows via the `declare` keyword, which allows developers to write function signatures and type declarations first and provide implementations later. Combined with the unified build/test/package workflow (`moon`), this yields a standardized end-to-end engineering workflow that closely matches real-world practice. Since SWE-AGI focuses on production-scale software systems that are largely absent from the current MoonBit ecosystem (e.g., a CDCL SAT solver, a WASM decoder/validator, and a standards-compliant C99 parser), it explicitly prioritizes *reasoning over retrieval*: success depends on sustained specification understanding, architectural decision-making, and disciplined long-horizon implementation rather than recalling near-matching reference code.

SWE-AGI targets production-scale software engineering and consists of 22 tasks spanning seven categories. These tasks are stratified into three difficulty tiers based on code volume and implementation complexity, comprising 6 easy, 8 medium, and 8 hard tasks. Completing the core logic of a SWE-AGI task requires 10^3–10^4 lines of implementation under a fixed API scaffold, corresponding to weeks to months of engineering effort for an experienced human developer. To support evaluation at this scale, each task provides normative specifications (`specs/`), an explicit task statement (`TASK.md`), and a visible public test subset for local iteration, while benchmark scoring is performed solely on final submissions evaluated against hidden private tests. This evaluation design shifts the challenge from isolated code generation to an end-to-end software engineering process, requiring agents to demonstrate sustained autonomy rather than relying on one-shot generation: interpreting complex specifications, becoming familiar with MoonBit, architecting modular systems, and performing self-directed testing.

In our latest evaluation, gpt-5.3-codex achieves the strongest overall performance (solving 19/22 tasks, 86.4%), outperforming gpt-5.2-codex (17/22, 77.3%), claude-opus-4.6 (15/22, 68.2%), and claude-opus-4.5 (10/22, 45.5%). Although these frontier agents successfully complete all easy-tier tasks, performance degrades on the medium and hard tiers as task difficulty increases: success rates for both gpt-5.3-codex and gpt-5.2-codex decline sharply on hard tasks, whereas claude-opus-4.6 and claude-opus-4.5 begin to falter from the medium tier onward. In addition, we evaluate several other LLMs on six easy-tier tasks, including gemini-3-flash, kimi-k2.5, claude-sonnet-4.5, deepseek-v3.2, glm-4.7, and qwen3-max. Most of these models solve at most 2/6 easy tasks, revealing a substantial performance gap relative to the evaluated frontier agents even at the lowest difficulty level. Among these easy-tier baselines, kimi-k2.5 achieves

[1] `https://github.com/moonbitlang/SWE-AGI`

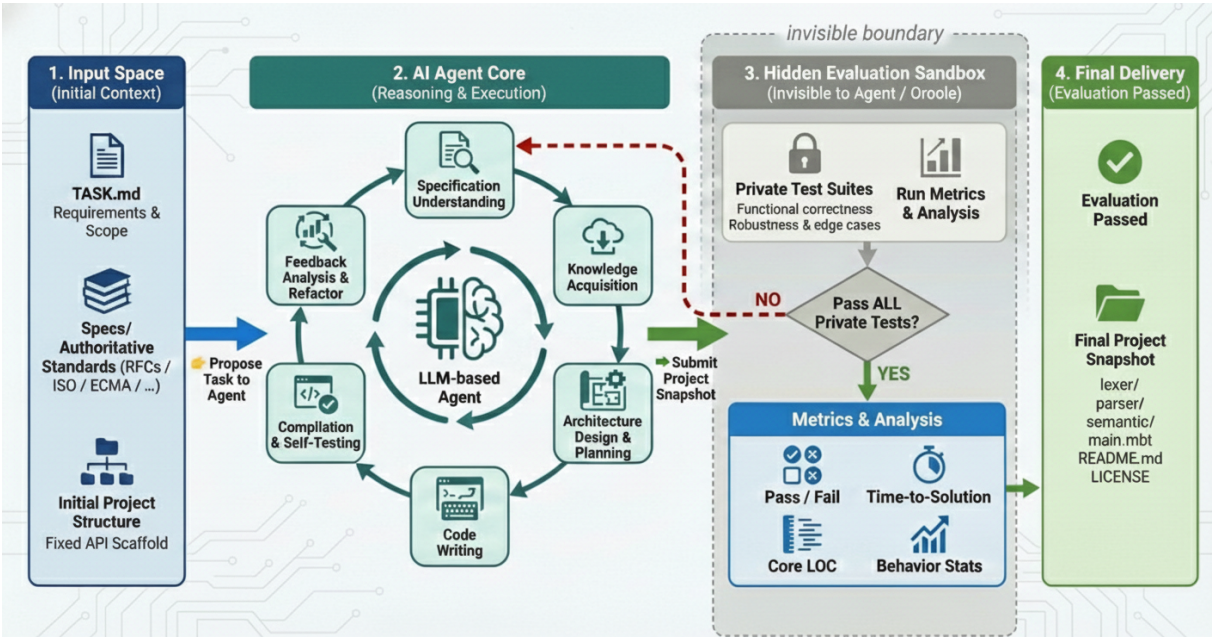
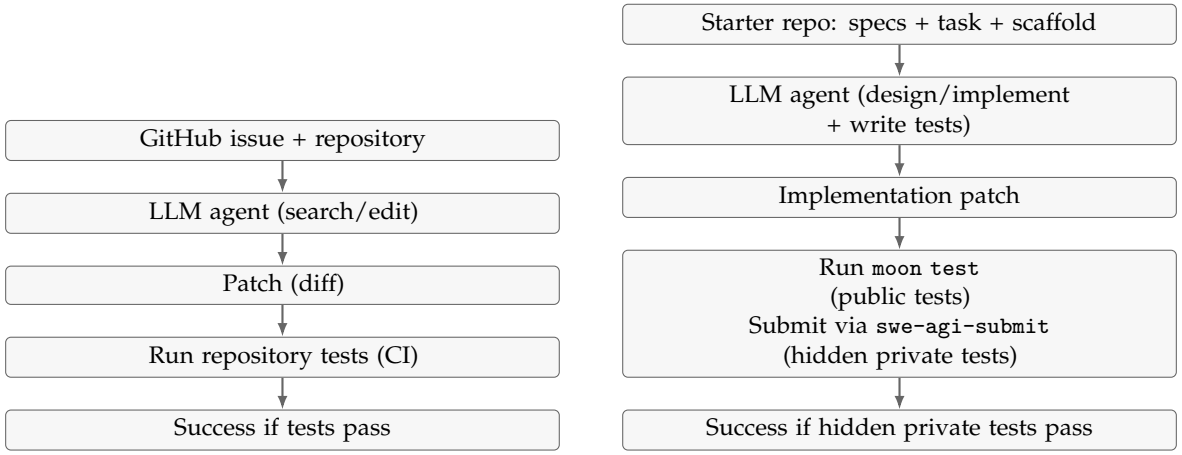


Figure 1 | SWE-AGI benchmark execution pipeline. From a cold-start starter repository (inputs: `TASK.md`, normative `specs/`, a MoonBit scaffold, and public tests), an autonomous agent iterates over design/implementation and local testing, submits the project for evaluation (via `swe-agi-submit`), receives pass/fail feedback, and repeats until a verified submission passes.

the highest overall test-suite pass rate (92.0%) while tying for the best task success rate (2/6). We further conduct a behavioral analysis of end-to-end SWE agents and observe that code reading, rather than code writing, emerges as the central bottleneck in AI-assisted software development. As codebases scale, maintaining a coherent modular architecture becomes the dominant activity. Consistent with this observation, gpt-5.2-codex allocates a larger fraction of its actions to code understanding, while gpt-5.3-codex exhibits a more iteration-oriented profile with higher debugging share and substantially fewer logged actions, improving time-to-solution and overall task completion efficiency. Overall, these results suggest that autonomous software engineering from explicit specifications is becoming increasingly feasible, yet remains far from a solved problem at production scale.

This paper makes three contributions:

- We introduce SWE-AGI, the first benchmark focusing on the end-to-end construction of complex systems from authoritative standards. It shifts the evaluation paradigm from localized code completion to long-horizon architectural reasoning and rigorous system implementation.
- We design a specification-grounded, retrieval-resistant evaluation setting by leveraging MoonBit's nascent ecosystem and spec-first primitives, ensuring that success reflects genuine long-horizon engineering capabilities rather than recall of near-matching artifacts.
- We benchmark state-of-the-art SWE agents built on frontier LLMs on SWE-AGI and present a comprehensive empirical and behavioral analysis, revealing strong performance on easy-tier tasks but substantial degradation as task difficulty increases.



(a) SWE-bench-style: issue resolution in existing repositories.

(b) SWE-AGI style: specification-driven implementation (with agent-written tests) in a fixed scaffold.

Figure 2 | Conceptual contrast between SWE-bench (Jimenez et al., 2023) and SWE-AGI evaluation settings.

2. SWE-AGI Benchmark

SWE-AGI evaluates autonomous software engineering through *specification-driven, from-scratch* construction of production-scale systems under a fixed MoonBit scaffold. Section 2.1 defines the per-task interface and agent execution loop, while Section 2.2 describes the benchmark construction process.

2.1. Task Formulation

Figure 1 illustrates the SWE-AGI execution pipeline. Each task is framed as the construction of a complete software system *from explicit specifications* (e.g., RFCs and standards) under a fixed MoonBit API scaffold. Concretely, a task is distributed as a starter repository that provides: (i) an explicit task statement (`TASK.md`) with acceptance criteria, constraints, and executable instructions; (ii) normative references (`specs/`); (iii) declaration-first API scaffolding that fixes the public interface; and (iv) a visible public test subset for fast local iteration. These components collectively define the core loop of the AI agent: interpreting the specifications, implementing against a fixed interface, validating locally, and iteratively submitting until the hidden private tests pass.

Evaluation considers only final submissions against hidden private tests, allowing agents full freedom in intermediate reasoning, testing, and implementation strategies. Private tests reduce overfitting to the visible suite and enforce specification-grounded implementations, while preserving an iterative, real-world-like engineering loop. During development, agents may supplement the provided public tests with their own spec-grounded checks, perform local validation via `moon test`, and iteratively submit solutions using `swe-agi-submit` until the submission passes the private test suite.

Figure 2 contrasts SWE-AGI with SWE-bench–style issue resolution in existing repositories. Compared to common coding benchmarks in Table 1, SWE-AGI shifts the primary sources of difficulty toward specification reading and operationalization, long-horizon system design and multi-module implementation, and iterative debugging/refactoring under build/test feedback in an open development setting[2]. Each task typically requires implementing 10^3–10^4 lines of

[2]External tools such as web search may be used, but are less helpful when near-matching implementations are

Table 1 | Comparison of SWE-AGI to representative coding and software engineering benchmarks (high-level characterization; code scale and workload are rough order-of-magnitude indicators).

Benchmark	Primary goal	Typical code scale	Workload	Difficulty focus	Evaluation criteria
HumanEval (Chen et al., 2021)	Function synthesis	$\sim 10^1$ LOC	minutes	Local correctness	Unit tests
MBPP (Austin et al., 2021)	Small programs	$\sim 10^1$–10^2 LOC	minutes–hours	Edge cases; basic reasoning	Unit tests
APPS (Hendrycks et al., 2021)	Programming problems	$\sim 10^2$–10^3 LOC	hours	Problem solving; I/O behavior	Test-based
LiveCodeBench (Jain et al., 2024)	Programming problems (time-based)	$\sim 10^2$–10^3 LOC	hours	Contamination-resistant coding skill	Test-based; time-evolving set
RepoBench (Liu et al., 2023b)	Repository-level completion	$\sim 10^1$–10^2 LOC	seconds–minutes	Cross-file context retrieval	Completion accuracy
SWE-bench (Jimenez et al., 2023)	Repo issue resolution	$\sim 10^1$–10^3 LOC	hours–days	Debugging; tool use; integration	Repository tests (CI)
SWE-bench Pro (Deng et al., 2025)	Repo issue resolution (enhanced)	$\sim 10^1$–10^3 LOC	hours–days	Debugging; improved coverage	Repository tests (CI)
SWE-AGI	Autonomous SWE from explicit specifications	$\sim 10^3$–10^4 LOC	weeks–months	Spec comprehension; system design	Hidden private tests via submission

core logic, corresponding to weeks or months of engineering effort for an experienced human developer, and is accompanied by high-coverage, human-validated test suites that evaluate both functional correctness on well-formed inputs and robustness to malformed inputs.

2.2. Benchmark Construction

SWE-AGI consists of 22 tasks spanning seven categories: (i) Template and Domain-Specific Languages (pug, jq); (ii) Data Serialization and Configuration Formats (csv, ini, yaml, toml); (iii) Markup and Document Formats (xml, html5); (iv) Programming Language Front-Ends (c99, lua, ecma262, python, r6rs); (v) Binary Formats and Streaming Decoders (git_object, protobuf, zip, capnp, wasm); (vi) Networking and Protocol State Machines (uri, hpack, url); and (vii) Automated Reasoning and SAT Solving (cdcl). Each task is framed as an end-to-end software system with a fixed API scaffold. Tasks are assigned to three coarse difficulty tiers (*Easy*/*Medium*/*Hard*), primarily based on the estimated scale of core implementation code (excluding tests), and further informed by semantic complexity indicators such as multi-phase parsing and validation, large state machines, and strict error-recovery requirements. Appendix A provides detailed task descriptions, per-task difficulty assignments, and overall tier counts.

SWE-AGI prioritizes *reasoning over retrieval* and is explicitly designed to minimize superficial success through memorization or direct code reuse. Accordingly, we focus on systems that are largely absent from the current MoonBit ecosystem and that demand sustained engagement

unavailable.

Listing 1 | Typical directory layout for a SWE-AGI task.

```
tasks/<task>/
  specs/                  # upstream specs and reference documents
  TASK.md                 # goal, scope, API, behavioral rules, test execution
  *_spec.mbt              # fixed API declarations + helper contracts
  *_pub_test.mbt          # public tests (subset of full suite)
  *_priv_test.mbt         # private tests (held out; only in evaluation
    checkout)
  moon.mod.json           # package manifest and dependencies
  moon.pkg.json           # package lockfile (pinned deps)
```

with formal specifications and non-trivial engineering decisions, including interface design, data-structure selection, and robust error handling.

Repository packaging. Following the interface defined in Section 2.1, tasks are constructed by selecting authoritative upstream specifications (e.g., standards, RFCs, and reference documents), distilling explicit acceptance criteria—including corner cases and error semantics—into `TASK.md`, and providing a fixed API scaffold together with high-coverage test suites. The test suites comprise a visible public subset for local iteration and a hidden private subset for verification. To support both agent usability and researcher auditability, each task directory includes normative references (`specs/`), a single task entry point (`TASK.md`), a minimal MoonBit package configuration (`moon.mod.json` and `moon.pkg.json`), and scaffolded declarations (typically in `*_spec.mbt`) that define and freeze the public API. Overall, tasks are packaged to minimize hidden requirements and evaluation variance, ensuring that success depends on specification-grounded engineering rather than repository-specific conventions. A typical directory layout is shown in Listing 1.

Test sets and evaluation metrics. Tests in SWE-AGI are constructed through a hybrid process. Canonical cases are adapted from authoritative specifications and reference materials, and are expanded with systematic edge cases—including property-based generators, LLM-generated candidates, and fuzz-style mutations where appropriate—followed by manual triage to ensure specification-consistent expectations. SWE-AGI reports both *functional* and *engineering* metrics (Table 2). Functional performance is measured by task success rate and test-suite pass rate (overall), while engineering effort and efficiency are characterized by time to solution and implementation size (core LOC), respectively. In addition, we report behavioral statistics to support more detailed analysis of agent behavior. Performance metrics such as runtime and memory usage are not scored in the current release, but are reserved for future versions once state-of-the-art models achieve consistently high task success rates.

2.3. Language Choice: MoonBit

SWE-AGI adopts MoonBit (MoonBit Team, 2025) as its implementation language to control distributional bias during evaluation. As a relatively new programming language with a still-maturing ecosystem, MoonBit is largely absent from existing large-scale pretraining corpora and public code repositories. This reduces the likelihood that agents can exploit memorized near-solutions or ecosystem-specific shortcuts, thereby shifting the evaluation signal toward specification comprehension, algorithmic reasoning, and correct end-to-end implementation.

MoonBit's type soundness and unified toolchain further improve the quality and timeliness

Table 2 | Recommended SWE-AGI metrics for reporting.

Metric	Definition
Task success rate	Fraction of tasks for which the final submission compiles successfully and passes all hidden private tests under the specified evaluation protocol.
Test-suite pass rate (overall)	Pass rate on the full evaluation test suite executed by the evaluator (public+private), reported as passed/total with no public/private split.
Time to solution	Wall-clock time to the first submission that passes the hidden private tests.
Implementation size (core LOC)	Number of non-test MoonBit lines of code, excluding public and private tests as well as auxiliary tooling, used as a coarse proxy for system scale.
Behavior stats (optional)	Aggregated distribution of agent actions over the engineering loop (e.g., specification reading, code reading and writing, debugging, test execution, planning or navigation, and external search), computed from tool usage logs.

```
declare pub(all) type CProgram
/// Parse a C99 translation unit from source text.
declare pub fn parse(code : StringView) -> CProgram raise
/// Encode the parsed program into the explicit test JSON schema
declare pub fn CProgram::to_test_json(self : CProgram) -> Json
```

Figure 3 | Declaration-first, spec-driven workflow in MoonBit. The `declare` keyword fixes public types and function signatures (e.g., parser entry points and test-schema encoders) before implementation.

of feedback available to autonomous agents. Its emphasis on data-oriented programming, immutability, and exhaustive pattern matching surfaces many classes of errors—such as missing cases, violated invariants, and type mismatches—at compile time rather than at runtime. Moreover, MoonBit implementations are often more concise for a given specification, reducing overall code volume and the surface area for latent bugs. Combined with fast compilation[3] and test execution via the `moon` toolchain, these properties enable high-frequency compile–test–refine cycles with low feedback latency, providing earlier and more actionable signals within the agent loop.

Finally, MoonBit's built-in support for separating interface and implementation enables a scaffolded evaluation setup in which public APIs, type signatures, and module boundaries are explicitly fixed using `declare` (Figure 3). Agents are required to implement the specified interfaces exactly, with deviations detected at compile time rather than implicitly tolerated at runtime. This enforces clear boundaries, prevents interface-level circumvention, and ensures that evaluation focuses on the correctness and robustness of the implemented logic rather than flexibility in interface design.

[3]In reported benchmarks, MoonBit can compile hundreds of packages in approximately one second, substantially reducing iteration overhead compared to traditional programming languages.

3. Evaluation of Frontier Agents

We evaluate software engineering agents built on frontier models on SWE-AGI under an open development setting in which the scored private tests are hidden from the model. Throughout, we use *model* to refer to the underlying LLM, and *agent* to refer to the model coupled with an execution front-end, tool access, and associated policies. Agents must translate `TASK.md` plus authoritative references (`specs/`) into a working MoonBit implementation under a fixed scaffold, iterate locally using public tests (10% of all tests), and submit via `swe-agi-submit` until the evaluator reports that hidden private tests pass.

3.1. Setup

We evaluate each model via an agent front-end that can edit the repository, execute local commands, and iteratively submit solutions. We use Codex CLI with gpt-5.3-codex and gpt-5.2-codex[4]; Gemini CLI with gemini-3-flash[5]; Claude Code with claude-opus-4.6, claude-opus-4.5, claude-sonnet-4.5, qwen3-max[6], glm-4.7, and deepseek-v3.2[7]; and Kimi CLI with kimi-k2.5[8]. We will release the execution scripts[9] along with the model outputs and corresponding run logs[10] to support reproducibility.

A task is considered *passed* if the final submitted project compiles and the evaluator reports zero failed hidden private tests in a clean checkout; otherwise it is *failed*. In addition to task-level success, we report test-suite pass rates (overall), wall-clock duration to the final submission (hours), implementation size (core LOC, excluding tests), and token usage aggregated from tool logs. We conduct full evaluations for gpt-5.3-codex, gpt-5.2-codex, claude-opus-4.6, and claude-opus-4.5. In addition, we conduct a rapid assessment of agentic coding capabilities on six easy-tier tasks for claude-sonnet-4.5, kimi-k2.5, glm-4.7, gemini-3-flash, deepseek-v3.2, and qwen3-max. Given the low easy-tier success rates, we limit these additional evaluations to the easy tier and do not extend testing to higher difficulty levels. We do not enforce an explicit budget constraint; instead, we report token consumption and wall-clock time as post hoc efficiency metrics aggregated per run from the recorded tool logs. For Claude Code executions (claude-opus-4.6 and claude-opus-4.5), we additionally report per-task monetary costs extracted from the agent logs in the detailed per-task tables.

3.2. Main Results

Overall performance. Table 3 summarizes SWE-AGI performance by difficulty tier and reveals a sharp difficulty gradient. On the easy tier, all evaluated frontier agents (gpt-5.3-codex, gpt-5.2-codex, claude-opus-4.6, claude-opus-4.5) solve 6/6 tasks with 100% test-suite pass rate, indicating that for small parsers/decoders the end-to-end loop (spec reading, implementation under a fixed scaffold, and iteration under test feedback) can be executed reliably. On the medium and hard tiers, outcomes diverge: gpt-5.3-codex solves 8/8 medium and 5/8 hard tasks

[4]For Codex CLI, we run gpt-5.3-codex in *xhigh* thinking mode. For gpt-5.2-codex, we adopt *high* thinking mode, since *xhigh* incurred prohibitively long wall-clock runtimes.

[5]In Gemini CLI runs, we observe repeated execution failures, including three instances of "Loop detected, stopping execution" and two instances of "[API Error: Premature close]", which resulted in a low task pass rate. Due to these stability issues, we omit results for gemini-3-pro from our reported evaluations.

[6]qwen3-max-thinking (2026-01-23)

[7]deepseek-reasoner

[8]kimi-k2.5-thinking

[9]`https://github.com/moonbitlang/SWE-AGI`

[10]`https://github.com/moonbitlang/SWE-AGI-Eval`

Table 3 | Evaluation summary by difficulty tier.

Difficulty	Model	Tasks Passed	Test-suite Pass Rate	Avg. Time	Avg. Core LOC
Easy	gpt-5.3-codex	**6/6**	**100.0% (avg of 6; total 1604/1604)**	0.28h	1305
	gpt-5.2-codex	**6/6**	**100.0% (avg of 6; total 1604/1604)**	0.81h	1081
	claude-opus-4.6	**6/6**	**100.0% (avg of 6; total 1604/1604)**	0.45h	1781
	claude-opus-4.5	**6/6**	**100.0% (avg of 6; total 1604/1604)**	0.39h	1092
	claude-sonnet-4.5	0/6	76.1% (avg of 6; total 556/1604)	0.32h	930
	kimi-k2.5	2/6	92.0% (avg of 6; total 1338/1604)	0.99h	1163
	glm-4.7	2/6	64.2% (avg of 6; total 456/1604)	0.70h	904
	gemini-3-flash	2/6	49.8% (avg of 6; total 376/1604)	0.25h	558
	deepseek-v3.2	1/6	16.7% (avg of 6; total 138/1604)	3.4h	1070
	qwen3-max	0/6	13.9% (avg of 6; total 94/1604)	2.6h	850
Medium	gpt-5.3-codex	**8/8**	**100.0% (avg of 8; total 5284/5284)**	1.2h	2575
	gpt-5.2-codex	**7/8**	**98.9% (avg of 8; total 5176/5284)**	5.1h	4702
	claude-opus-4.6	5/8	93.6% (avg of 8; total 5146/5284)	3.5h	4867
	claude-opus-4.5	3/8	82.6% (avg of 8; total 4593/5284)	1.3h	3304
Hard	gpt-5.3-codex	**5/8**	**87.9% (avg of 8; total 13976/15638)**	1.7h	6255
	gpt-5.2-codex	**4/8**	**91.2% (avg of 8; total 13205/15638)**	7.8h	9034
	claude-opus-4.6	4/8	81.1% (avg of 8; total 14625/15638)	5.7h	10103
	claude-opus-4.5	1/8	67.0% (avg of 8; total 10621/15638)	1.7h	6603

(19/22 overall), gpt-5.2-codex solves 7/8 medium and 4/8 hard (17/22), claude-opus-4.6 solves 5/8 medium and 4/8 hard (15/22), while claude-opus-4.5 solves 3/8 medium and 1/8 hard (10/22). This widening separation suggests that scaling to larger, more specification-intensive systems is the key differentiator among frontier agents in SWE-AGI.

We also run a rapid easy-tier sweep of additional models. Even within this easier regime, success rates are low. kimi-k2.5, glm-4.7, and gemini-3-flash solve only 2/6 tasks. deepseek-v3.2 solves 1/6, while claude-sonnet-4.5 and qwen3-max solve 0/6. These results indicate that SWE-AGI is sensitive to robustness and generalization under specification pressure: models that appear close on code-centric open benchmarks can separate substantially once placed in an end-to-end setting with hidden private tests.

Failure to solve a task does not always indicate broad functional incorrectness. Across tiers, many "failed" submissions still pass a large fraction of the evaluation test suite, suggesting that remaining defects are often localized to rare normative requirements, subtle state-machine corner cases, or performance bottlenecks that only surface in the hidden private tests. This is most pronounced on the hard tier: despite solving fewer hard tasks than gpt-5.3-codex (4/8 vs. 5/8), gpt-5.2-codex achieves a higher unweighted mean hard-tier test-suite pass rate (91.2%), reflecting near-complete coverage on several failures. At the task level, we observe multiple near-misses, e.g., `cdcl` reaches 99.8% test-suite pass rate for gpt-5.2-codex and `lua` reaches 96.4% for claude-opus-4.5 (Tables 4 and 5). Practically, this means the pass/fail boundary is often dominated by eliminating the last few spec-sensitive edge cases rather than constructing missing core subsystems.

Agent Efficiency. Average wall-clock time and code size primarily reflect long-horizon engineering difficulty and agent efficiency bottlenecks, rather than pure model capability; both are also strongly influenced by the chosen front-end configuration and tool policies, and should therefore be interpreted with caution. Within this framing, Table 3 highlights two consistent gaps. First, gpt-5.3-codex is substantially more time-efficient than gpt-5.2-codex while also improving task completion: its average runtime is about 3–5× lower across tiers (0.28h vs. 0.81h

Table 4 | Per-task detailed results for gpt-5.3-codex and gpt-5.2-codex. Tokens report input/output tokens as logged; for Codex CLI we report *input_tokens* (excluding *cached_input_tokens*). Cost reports per-task dollar cost; values for Codex CLI are approximate (using API price), and this estimate is inaccurate since it ignores the overhead introduced by reasoning tokens. Due to the excessively long runtime of *ecma262* (exceeding 42 hours), we evaluate it using a 42-hour snapshot. As the execution did not finish within this window, input/output token statistics are unavailable in the logs and are reported as *N/A*.

Task	Model	Task Passed	Test-suite Pass Rate	Duration	Core LOC	Tokens (in/out)	Cost ($)
`pug`	gpt-5.3-codex	Yes	**100.0% (251/251)**	3.5h	2709	92.73M/319.3k	~22.69
	gpt-5.2-codex	Yes	**100.0% (251/251)**	24.6h	14251	647.72M/3.28M	~176.90
`jq`	gpt-5.3-codex	Yes	**100.0% (218/218)**	1.1h	4914	45.68M/184.9k	~11.41
	gpt-5.2-codex	Yes	**100.0% (218/218)**	1.9h	6416	68.93M/267.2k	~16.67
`csv`	gpt-5.3-codex	Yes	**100.0% (98/98)**	0.21h	467	5.79M/37.1k	~1.85
	gpt-5.2-codex	Yes	**100.0% (98/98)**	0.68h	440	12.97M/78.9k	~3.69
`ini`	gpt-5.3-codex	Yes	**100.0% (98/98)**	0.26h	1495	9.16M/46.9k	~2.57
	gpt-5.2-codex	Yes	**100.0% (98/98)**	0.77h	927	17.29M/104.7k	~4.95
`yaml`	gpt-5.3-codex	Yes	**100.0% (345/345)**	0.90h	856	26.04M/148.3k	~47.64
	gpt-5.2-codex	Yes	**100.0% (345/345)**	3.9h	3664	95.40M/571.6k	~26.78
`toml`	gpt-5.3-codex	Yes	**100.0% (733/733)**	0.88h	2149	20.80M/153.2k	~6.64
	gpt-5.2-codex	Yes	**100.0% (733/733)**	3.0h	2280	64.37M/345.0k	~17.25
`xml`	gpt-5.3-codex	Yes	**100.0% (735/735)**	1.2h	4504	40.93M/165.3k	~10.57
	gpt-5.2-codex	Yes	**100.0% (735/735)**	1.9h	4946	61.76M/230.3k	~14.81
`html5`	gpt-5.3-codex	No	**86.2% (7086/8221)**	3.3h	11080	67.40M/155.5k	~15.45
	gpt-5.2-codex	No	78.4% (6444/8221)	3.0h	6433	51.76M/209.1k	~12.66
`c99`	gpt-5.3-codex	Yes	**100.0% (117/117)**	0.62h	3624	22.06M/99.3k	~5.82
	gpt-5.2-codex	Yes	**100.0% (117/117)**	3.1h	5052	66.09M/332.6k	~17.68
`lua`	gpt-5.3-codex	Yes	**100.0% (137/137)**	0.93h	9625	22.97M/163.2k	~7.09
	gpt-5.2-codex	Yes	**100.0% (137/137)**	2.5h	5574	71.14M/307.1k	~18.40
`ecma262`	gpt-5.3-codex	No	17.5% (108/618)	0.75h	7445	17.92M/109.6k	~5.45
	gpt-5.2-codex	No	**97.7% (604/618)**	42.2h	33302	N/A	N/A
`python`	gpt-5.3-codex	Yes	**100.0% (653/653)**	2.3h	7953	51.54M/228.4k	~13.86
	gpt-5.2-codex	Yes	**100.0% (653/653)**	1.8h	7675	64.68M/236.8k	~17.70
`r6rs`	gpt-5.3-codex	Yes	**100.0% (1362/1362)**	2.6h	3751	52.46M/225.0k	~13.72
	gpt-5.2-codex	No	53.4% (727/1362)	2.9h	6436	78.02M/349.6k	~20.19
`git_object`	gpt-5.3-codex	Yes	**100.0% (1000/1000)**	0.44h	840	9.85M/42.4k	~2.80
	gpt-5.2-codex	Yes	**100.0% (1000/1000)**	1.2h	1164	37.96M/148.5k	~9.78
`protobuf`	gpt-5.3-codex	Yes	**100.0% (141/141)**	0.19h	1590	7.09M/34.4k	~2.05
	gpt-5.2-codex	Yes	**100.0% (141/141)**	0.50h	1670	14.26M/74.5k	~3.83
`zip`	gpt-5.3-codex	Yes	**100.0% (1089/1089)**	1.2h	1258	41.19M/177.0k	~10.83
	gpt-5.2-codex	Yes	**100.0% (1089/1089)**	3.2h	1346	77.73M/373.3k	~20.32
`capnp`	gpt-5.3-codex	Yes	**100.0% (111/111)**	0.53h	3114	17.83M/82.4k	~4.88
	gpt-5.2-codex	Yes	**100.0% (111/111)**	1.2h	2798	25.16M/136.3k	~7.04
`wasm`	gpt-5.3-codex	Yes	**100.0% (800/800)**	0.76h	5085	29.51M/125.1k	~8.01
	gpt-5.2-codex	Yes	**100.0% (800/800)**	2.1h	3479	59.27M/270.4k	~15.67
`uri`	gpt-5.3-codex	Yes	**100.0% (138/138)**	0.25h	1645	8.98M/45.2k	~2.63
	gpt-5.2-codex	Yes	**100.0% (138/138)**	0.72h	1128	15.90M/101.3k	~4.59
`hpack`	gpt-5.3-codex	Yes	**100.0% (129/129)**	0.33h	1793	11.71M/56.7k	~3.10
	gpt-5.2-codex	Yes	**100.0% (129/129)**	1.0h	1157	23.90M/143.9k	~6.68
`url`	gpt-5.3-codex	Yes	**100.0% (1220/1220)**	0.32h	926	11.36M/48.5k	~2.88
	gpt-5.2-codex	No	91.1% (1112/1220)	1.2h	4849	31.86M/159.2k	~8.40
`cdcl`	gpt-5.3-codex	No	99.6% (4295/4312)	2.2h	1650	53.07M/96.7k	~11.14
	gpt-5.2-codex	No	**99.8% (4305/4312)**	5.1h	1380	47.34M/181.8k	~11.75

Table 5 | Per-task detailed results for claude-opus-4.6 and claude-opus-4.5. Token and cost values are extracted from Claude Code logs when available; *N/A* indicates missing token/cost logs (e.g., due to a Claude Code crash: *Maximum call stack size exceeded*).

Task	Model	Task Passed	Test-suite Pass Rate	Duration	Core LOC	Tokens (in/out)	Cost ($)
`pug`	claude-opus-4.6	No	51.4% (129/251)	1.8h	4246	44.57M/138.2k	51.69
	claude-opus-4.5	No	35.5% (89/251)	1.1h	3422	30.96M/130.8k	22.82
`jq`	claude-opus-4.6	Yes	**100.0% (218/218)**	1.3h	6055	27.74M/170.3k	25.49
	claude-opus-4.5	Yes	**100.0% (218/218)**	1.3h	7812	48.46M/245.9k	34.47
`csv`	claude-opus-4.6	Yes	**100.0% (98/98)**	0.70h	474	16.27M/91.7k	12.44
	claude-opus-4.5	Yes	**100.0% (98/98)**	0.49h	483	15.54M/65.7k	11.89
`ini`	claude-opus-4.6	Yes	**100.0% (98/98)**	0.54h	923	8.94M/80.7k	9.05
	claude-opus-4.5	Yes	**100.0% (98/98)**	0.28h	1070	8.28M/53.9k	6.43
`yaml`	claude-opus-4.6	No	99.7% (344/345)	1.8h	3721	42.91M/142.0k	29.83
	claude-opus-4.5	No	68.1% (235/345)	1.4h	3596	46.98M/207.7k	33.91
`toml`	claude-opus-4.6	No	98.0% (718/733)	13.0h	6779	N/A	N/A
	claude-opus-4.5	No	82.3% (603/733)	0.65h	2855	26.86M/106.9k	18.27
`xml`	claude-opus-4.6	Yes	**100.0% (735/735)**	1.6h	2839	50.70M/172.3k	35.88
	claude-opus-4.5	Yes	**100.0% (735/735)**	2.4h	3241	58.68M/207.2k	39.42
`html5`	claude-opus-4.6	Yes	**100.0% (8221/8221)**	12.5h	10585	N/A	N/A
	claude-opus-4.5	No	56.5% (4648/8221)	1.0h	7583	17.39M/110.0k	14.38
`c99`	claude-opus-4.6	Yes	**100.0% (117/117)**	1.1h	4979	23.48M/100.5k	27.12
	claude-opus-4.5	No	45.3% (53/117)	0.69h	4937	30.44M/96.2k	20.39
`lua`	claude-opus-4.6	No	97.1% (133/137)	1.5h	6688	43.37M/155.4k	398.24
	claude-opus-4.5	No	96.4% (132/137)	1.2h	6782	46.14M/220.0k	34.15
`ecma262`	claude-opus-4.6	No	60.2% (372/618)	9.3h	13901	N/A	N/A
	claude-opus-4.5	No	23.1% (143/618)	0.64h	5172	34.13M/107.6k	21.78
`python`	claude-opus-4.6	No	0.0% (0/653)	5.2h	16991	127.99M/365.6k	135.91
	claude-opus-4.5	No	60.8% (397/653)	3.2h	10932	84.85M/252.1k	60.45
`r6rs`	claude-opus-4.6	No	91.9% (1252/1362)	7.9h	20422	21.28M/199.5k	190.32
	claude-opus-4.5	No	54.0% (735/1362)	2.9h	8585	51.90M/214.4k	36.57
`git_object`	claude-opus-4.6	Yes	**100.0% (1000/1000)**	0.36h	1291	6.79M/40.5k	8.27
	claude-opus-4.5	Yes	**100.0% (1000/1000)**	0.68h	1065	23.93M/66.9k	15.81
`protobuf`	claude-opus-4.6	Yes	**100.0% (141/141)**	0.20h	858	6.45M/28.0k	4.77
	claude-opus-4.5	Yes	**100.0% (141/141)**	0.17h	1051	6.70M/32.4k	5.21
`zip`	claude-opus-4.6	Yes	**100.0% (1089/1089)**	8.1h	10530	58.10M/227.1k	1016.53
	claude-opus-4.5	No	88.0% (958/1089)	2.1h	2006	56.33M/174.9k	37.19
`capnp`	claude-opus-4.6	Yes	**100.0% (111/111)**	0.38h	2605	10.63M/57.8k	9.37
	claude-opus-4.5	Yes	**100.0% (111/111)**	0.25h	2690	7.09M/47.6k	7.55
`wasm`	claude-opus-4.6	Yes	**100.0% (800/800)**	0.54h	4152	16.50M/92.3k	13.66
	claude-opus-4.5	Yes	**100.0% (800/800)**	1.7h	6101	44.49M/164.6k	30.91
`uri`	claude-opus-4.6	Yes	**100.0% (138/138)**	0.23h	1198	5.93M/39.6k	4.72
	claude-opus-4.5	Yes	**100.0% (138/138)**	0.31h	1320	8.01M/56.7k	6.81
`hpack`	claude-opus-4.6	Yes	**100.0% (129/129)**	0.68h	5941	9.37M/63.3k	9.37
	claude-opus-4.5	Yes	**100.0% (129/129)**	0.38h	1561	10.84M/75.2k	10.53
`url`	claude-opus-4.6	Yes	**100.0% (1220/1220)**	1.2h	4065	29.16M/126.3k	26.97
	claude-opus-4.5	No	87.0% (1062/1220)	1.1h	2517	27.95M/94.5k	18.36
`cdcl`	claude-opus-4.6	Yes	**100.0% (4312/4312)**	6.7h	1203	60.77M/221.0k	46.19
	claude-opus-4.5	No	99.6% (4295/4312)	3.0h	1020	31.56M/82.9k	20.63

on easy, 1.2h vs. 5.1h on medium, 1.7h vs. 7.8h on hard), and its average implementations are smaller on medium and hard tasks (2575 vs. 4702 core LOC on medium; 6255 vs. 9034 on hard). Second, claude-opus-4.6 improves substantially over claude-opus-4.5 on medium and hard tiers (15/22 vs. 10/22 overall), but this gain comes with higher wall-clock time on those tiers (3.5h vs. 1.3h on medium; 5.7h vs. 1.7h on hard), consistent with additional exploration and debugging under specification pressure.

At the same time, the runs reveal a noteworthy capability of gpt-5.2-codex: sustained long-horizon execution even when convergence fails. For example, on `ecma262` the agent runs for 42 hours without early termination while still failing the private test suite, producing an unusually large implementation (over 30k core LOC). Accordingly, we treat core LOC as a coarse indicator of implementation scale rather than an optimization target: higher LOC may indicate broader feature coverage, but may also reflect verbose implementations and refactoring churn under heavy specification pressure.

3.3. End-to-End SWE Behavior Analysis

Beyond pass/fail outcomes, we analyze how agents allocate effort over long trajectories by labeling logged tool actions into coarse SWE-relevant behavior categories. The taxonomy (Table 6) is heuristic: it maps observable actions (shell commands, file reads/writes, test runs, submissions, etc.) to a small set of intent-level buckets that approximate the engineering loop (spec understanding, code understanding/writing, debugging, hygiene, and external search). These statistics do not capture unlogged internal reasoning, and absolute counts depend on each agent front-end's logging granularity; we therefore interpret them as qualitative indicators of *effort allocation* rather than a normalized efficiency metric.

Table 7 summarizes the distribution of agent behaviors across difficulty tiers. As difficulty increases, code understanding (Read) becomes the dominant activity and interaction volume grows sharply for several agents. On hard tasks, Read accounts for 41.4% of logged actions for gpt-5.3-codex and 64.6% for gpt-5.2-codex, with claude-opus-4.6 at 50.2% and claude-opus-4.5 at 43.5%. This shift coincides with a large increase in total actions: on hard tasks, gpt-5.2-codex averages 1676 logged actions per task, compared to 301 for gpt-5.3-codex and 1498 for claude-opus-4.6. Overall, once implementations reach multi-module, spec-heavy regimes, agents devote a substantial fraction of their effort to reading, inspecting, and validating existing code rather than generating new functionality.

These patterns suggest that long-horizon progress is constrained less by raw code generation capacity than by the ability to maintain and reason over an evolving codebase. In this setting the bottleneck shifts toward preserving architectural consistency, understanding prior design decisions, and verifying interactions across modules. This aligns with findings in Thomas (2026) that identify code reading—rather than code writing—as a central bottleneck in AI-assisted software development, and supports the view that comprehension and maintenance costs dominate long-horizon engineering.

Strategy Differences Across Frontier Agents. Frontier agents exhibit systematic differences in workflow that track within-family improvements. Relative to gpt-5.2-codex, gpt-5.3-codex is markedly more iteration-oriented on medium and hard tasks: it spends a smaller share on Read (41.4% vs. 64.6% on hard) while allocating more to Debug (19.8% vs. 9.2%), and it completes runs with far fewer logged actions (301 vs. 1676 on hard). This profile is consistent with faster convergence: fewer prolonged "maintenance" phases dominated by reading and

Table 6 | SWE behavior categories used for log-based analysis. Categories are heuristic labels applied to logged tool actions to summarize effort allocation.

Abbrev.	Category	Definition (typical signals)
Action	Action count	Counted logged tool actions for a task run (used as a coarse proxy for interaction volume).
Spec	Spec understanding	Reading/searching requirements and expected behavior (e.g., `TASK.md`, `specs/`, public tests).
Plan	Planning	Creating or updating task-level plans/todo items.
Read	Code understanding	Reading/searching implementation code or inspecting artifacts to understand implementation; includes repository exploration and file navigation (e.g., `ls`, `find`, help).
Write	Code writing	Creating/modifying project files (implementation/config), including edits and file operations.
Debug	Debugging	Running builds/tests/submissions and investigating failures; includes non-zero-exit actions.
Hyg	Hygiene	Formatting or mechanical refactors explicitly recorded (e.g., `moon fmt`).
Ext	External search	Fetching/searching resources outside the repository.
Other	Other	Remaining actions that do not clearly fit the above categories.

Table 7 | Behavior summary by difficulty tier (percent of logged actions). Action reports average counted actions per task. Top-3 behavior shares per row are bold.

Difficulty	Model	Action	Spec	Plan	Read	Write	Debug	Hyg	Ext	Other
Easy	gpt-5.3-codex	109	9.3%	0.2%	**46.0%**	**24.8%**	**17.7%**	0.5%	0.0%	1.5%
	gpt-5.2-codex	195	6.8%	0.4%	**50.1%**	**25.3%**	**12.1%**	3.4%	0.0%	1.9%
	claude-opus-4.6	150	11.9%	6.6%	**35.1%**	**24.1%**	**17.4%**	0.3%	1.2%	3.3%
	claude-opus-4.5	145	14.9%	6.6%	**25.3%**	**27.0%**	**24.0%**	0.1%	0.3%	1.8%
	claude-sonnet-4.5	165	5.3%	5.5%	**25.8%**	**35.9%**	**24.1%**	0.4%	0.0%	3.1%
	kimi-k2.5	138	6.7%	3.3%	**20.3%**	**26.8%**	**33.7%**	0.5%	0.0%	8.8%
	glm-4.7	232	4.7%	3.7%	**30.8%**	**26.3%**	**27.9%**	0.1%	1.1%	5.3%
	gemini-3-flash	79	5.9%	0.0%	**12.9%**	**46.2%**	**33.7%**	0.0%	0.2%	1.1%
	deepseek-v3.2	608	4.2%	3.3%	**36.2%**	**38.4%**	**17.1%**	0.1%	0.2%	0.4%
	qwen3-max	198	7.1%	4.4%	**24.5%**	**42.4%**	**21.2%**	0.0%	0.4%	0.1%
Medium	gpt-5.3-codex	311	7.2%	0.0%	**39.6%**	**19.3%**	**19.1%**	0.8%	6.0%	7.9%
	gpt-5.2-codex	1070	6.1%	0.0%	**50.1%**	**21.1%**	**11.6%**	1.9%	2.0%	7.2%
	claude-opus-4.6	938	8.2%	5.7%	**43.1%**	**13.8%**	**14.2%**	0.1%	10.2%	4.6%
	claude-opus-4.5	381	6.2%	3.6%	**35.9%**	**25.8%**	**24.9%**	0.2%	0.2%	3.1%
Hard	gpt-5.3-codex	301	6.0%	0.0%	**41.4%**	**20.9%**	**19.8%**	0.5%	2.0%	9.3%
	gpt-5.2-codex	1676	2.3%	0.0%	**64.6%**	**20.5%**	**9.2%**	1.0%	0.2%	2.1%
	claude-opus-4.6	1498	6.6%	6.7%	**50.2%**	**13.3%**	**16.2%**	0.1%	3.4%	3.5%
	claude-opus-4.5	434	5.2%	4.4%	**43.5%**	**24.5%**	**20.3%**	0.1%	0.2%	1.8%

more decisive test–fix–retest loops, yielding substantially lower wall-clock time while improving task completion (Table 3).

Within the Claude family, claude-opus-4.6 improves substantially over claude-opus-4.5 on medium and hard tiers, and its behavior suggests a more deliberate workflow. Compared to claude-opus-4.5, it allocates more effort to specification engagement and planning (e.g., on hard tasks: 6.6% Spec and 6.7% Plan vs. 5.2% Spec and 4.4% Plan) and less to raw code writing (13.3% vs. 24.5%), while maintaining a comparable debugging share (16.2% vs. 20.3%). This shift toward reading and planning appears beneficial on spec-heavy systems where naive patching can destabilize global invariants. In contrast, claude-opus-4.5 exhibits a more pronounced "read specification–patch–rerun" pattern, with higher Write and Debug shares across tiers. While such a strategy can be effective on smaller tasks where localized fixes converge quickly, on

complex state-machine–driven systems (e.g., the HTML5 parser) frequent local patches may accumulate inconsistencies and degrade architectural coherence, leading to instability rather than convergence.

4. Related Work

Evaluation of LLMs. Broad evaluation frameworks such as HELM (Liang et al., 2022) and BIG-bench (Srivastava et al., 2022) emphasize multi-scenario, multi-metric measurement, highlighting trade-offs beyond accuracy, such as robustness and efficiency. As LLMs increasingly transition into autonomous agents (Schick et al., 2023; Shinn et al., 2023; Yao et al., 2022), evaluation has shifted from static prompting to interactive environments that stress tool use, multi-step planning, and long-horizon consistency. While domain-agnostic benchmarks like AgentBench (Liu et al., 2023c) and Terminal-Bench (The Terminal-Bench Team, 2025) provide foundational infrastructure, SWE-AGI focuses on the unique constraints of software engineering. It departs from the repository-centric paradigm of SWE-bench (Jimenez et al., 2023) in two key ways: (i) tasks are defined by rigorous, ground-truth specifications rather than existing codebase conventions, and (ii) it employs a submission-based sandbox with private, non-public test suites, ensuring auditable measurement even for models with unrestricted web search and retrieval capabilities.

Software Engineering Benchmarks. The evaluation of code intelligence has evolved from snippet-level synthesis to full-lifecycle engineering. Early benchmarks like HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) focus on isolated function-level tasks, while efforts like EvalPlus (Liu et al., 2023a) address test-case insufficiency. To counter data contamination, LiveCodeBench (Jain et al., 2024) introduced continuous curation. However, real-world engineering requires reasoning across multiple files, as explored in RepoBench (Liu et al., 2023b) and SWE-bench (Jimenez et al., 2023). Recently, the design space has expanded toward specialized dimensions: PRDBench (Fu et al., 2025) targets PRD-to-code workflows; OSS-Bench (Jiang et al., 2025) focuses on memory-safety and optimization; and SWE-EVO (Thai et al., 2025) shifts from initial construction to continuous software evolution. SWE-AGI complements this landscape by targeting the end-to-end systems regime: agents must build a complete, robust system from high-level specs under a fixed API. By decoupling the evaluation from visible unit tests and existing repository noise, SWE-AGI provides a cleaner signal for an agent's ability to handle the "requirements-to-implementation" gap—a critical frontier for production-scale AI engineering.

Programming Languages and LLMs. Programming languages and ecosystems shape what models can learn and how reliably they generalize. MultiPL-E (Cassano et al., 2022) shows that model performance and failure modes vary across languages, reflecting differences in syntax, standard libraries, tooling, and conventions. Beyond syntax, effective AI coding increasingly depends on a "full-stack" tool-and-feedback loop: editor/refactoring support, build systems, test runners, linters, static analyzers, profilers, and submission/evaluation harnesses that provide fast and accurate signals. In many real deployments, the bottleneck is not code generation but review, debugging, integration, and specification clarification—suggesting an advantage for languages and platforms that shift feedback from humans to machines via strong static guarantees, deterministic builds, and rich automated checks.

This favors statically typed languages and ecosystems that integrate a one-stop toolchain and enforce disciplined interfaces, enabling agents to iterate with high-quality feedback and

fewer ambiguous failure modes. As the fraction of AI-generated code grows, language and platform design may increasingly optimize for machine-assisted development: explicit specifications, stable API scaffolds, auditable build/test pipelines, and standard diagnostics that can be consumed by agents. SWE-AGI uses MoonBit (MoonBit Team, 2025), a recently developed programming language with an integrated toolchain: the `declare` keyword supports declaration-first scaffolding under a fixed API, and the unified workflow (`moon`) supports fast compilation, reproducible builds, and submission-style evaluation at production scale.

5. Conclusion

SWE-AGI evaluates LLM-based software engineering agents on tasks defined by explicit specifications and measured by deterministic, human-validated tests. The benchmark targets production-quality, from-scratch MoonBit implementations in the 10^3–10^4 LOC regime and is evaluated through an iterative submission protocol: agents build and test locally, submit via `swe-agi-submit`, and receive pass/fail feedback from hidden private tests. Across 22 tasks spanning seven specification families, we observe a steep difficulty gradient: frontier agents reliably solve all easy tasks, but performance drops sharply on medium and hard tiers. Overall, gpt-5.3-codex solves 19/22 tasks (86.4%), gpt-5.2-codex solves 17/22 (77.3%), claude-opus-4.6 solves 15/22 (68.2%), and claude-opus-4.5 solves 10/22 (45.5%). Many failures are near-misses with high test-suite pass rates, suggesting that the pass/fail boundary is often dominated by a small number of specification-sensitive edge cases and performance corner cases rather than missing major subsystems.

Complementing these outcome metrics, our log-based behavior analysis indicates that long-horizon progress is increasingly dominated by code understanding and maintenance rather than raw code writing. As difficulty increases, agents spend a growing share of actions reading and inspecting evolving implementations, and systematic differences in Read/Write/Debug allocation track within-family performance improvements. These findings reinforce that the central bottleneck in end-to-end agentic software engineering is sustaining coherent, correct systems over long trajectories under build/test feedback.

In future work, we will extend SWE-AGI to encompass heterogeneous distributed systems and complex legacy code integration tasks that demand deep architectural reasoning. We also plan to study library-centric workflows: how agents decompose specifications into reusable components, divide subtasks across libraries, and compose existing libraries into even larger software systems. Finally, incorporating multi-modal inputs (e.g., architectural diagrams and visual execution traces) and exploring agent-centric toolchain optimizations alongside non-functional imperatives like security and maintainability will be essential for achieving deterministic, production-grade reliability.

References

Anthropic. Claude sonnet 4.5. `https://www.anthropic.com/news/claude-sonnet-4-5`, 2025.

J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021. URL `https://arxiv.org/abs/2108.07732`.

F. Cassano, J. Gouwar, D. Nguyen, et al. MultiPL-E: A scalable and extensible approach to benchmarking neural code generation, 2022. URL `https://arxiv.org/abs/2208.08227`.

M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, J. Hilton, R. Nakano, C. Hesse, J. Chen, E. Sigler, D. Ziegler, N. Stiennon,

J. Wu, A. Radford, D. Amodei, and I. Sutskever. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021. URL https://arxiv.org/abs/2107.03374.

DeepSeek Team et al. Deepseek-v3.2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*, 2025. URL https://arxiv.org/abs/2512.02556.

X. Deng, J. Da, E. Pan, et al. SWE-Bench pro: Can AI agents solve long-horizon software engineering tasks?, 2025. URL https://arxiv.org/abs/2509.16941.

L. Fu, B. Zhang, H. Guan, Y. Zhu, L. Qiu, W. Liu, X. Cao, X. Cai, W. Zhang, and Y. Yu. Automatically benchmarking llm code agents through agent-driven annotation and evaluation, 2025. URL https://arxiv.org/abs/2510.24358.

Gemini Team et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025. URL https://arxiv.org/abs/2507.06261.

D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song, and J. Steinhardt. Measuring coding challenge competence with APPS. *arXiv preprint arXiv:2105.09938*, 2021. URL https://arxiv.org/abs/2105.09938.

N. Jain, K. Han, A. Gu, W.-D. Li, F. Yan, T. Zhang, S. Wang, A. Solar-Lezama, K. Sen, and I. Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024. URL https://arxiv.org/abs/2403.07974.

Y. Jiang, R. Yap, and Z. Liang. Oss-bench: Benchmark generator for coding llms, 2025. URL https://arxiv.org/abs/2505.12331.

C. E. Jimenez, J. Yang, A. Wettig, et al. SWE-bench: Can language models resolve real-world GitHub issues? *arXiv preprint arXiv:2310.06770*, 2023. URL https://arxiv.org/abs/2310.06770.

Kimi Team et al. Kimi k2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*, 2025. URL https://arxiv.org/abs/2507.20534.

P. Liang, R. Bommasani, T. Lee, D. Tsipras, D. Soylu, M. Yasunaga, Y. Zhang, D. Narayanan, Y. Wu, A. Kumar, et al. Holistic evaluation of language models. *arXiv preprint arXiv:2211.09110*, 2022. URL https://arxiv.org/abs/2211.09110.

J. Liu, C. S. Xia, Y. Wang, and L. Zhang. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation, 2023a. URL https://arxiv.org/abs/2305.01210.

T. Liu, C. Xu, and J. McAuley. RepoBench: Benchmarking repository-level code auto-completion systems. *arXiv preprint arXiv:2306.03091*, 2023b. URL https://arxiv.org/abs/2306.03091.

X. Liu, H. Yu, H. Zhang, Y. Xu, X. Lei, H. Lai, Y. Gu, H. Ding, K. Men, K. Yang, S. Zhang, X. Deng, A. Zeng, Z. Du, C. Zhang, S. Shen, T. Zhang, Y. Su, H. Sun, M. Huang, Y. Dong, and J. Tang. Agentbench: Evaluating LLMs as agents, 2023c. URL https://arxiv.org/abs/2308.03688.

MoonBit Team. MoonBit programming language. https://www.moonbitlang.com/, 2025.

OpenAI. OpenAI GPT-5 system card. *arXiv preprint arXiv:2601.03267*, 2025. URL https://arxiv.org/abs/2601.03267.

Qwen Team et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025. URL https://arxiv.org/abs/2505.09388.

T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, L. Zettlemoyer, and N. Cancedda. Toolformer: Language models can teach themselves to use tools, 2023. URL https://arxiv.org/abs/2302.04761.

N. Shinn, B. Labash, A. Gopinath, K. Narasimhan, and S. Yao. Reflexion: Language agents with verbal reinforcement learning, 2023. URL https://arxiv.org/abs/2303.11366.

A. Srivastava, A. Rastogi, A. Rao, A. A. M. Shoeb, A. Abid, A. Fisch, A. R. Brown, A. Santoro, A. Gupta, A. Garriga-Alonso, et al. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *arXiv preprint arXiv:2206.04615*, 2022. URL https://arxiv.org/abs/2206.04615.

M. V. T. Thai, T. Le, D. N. Manh, H. P. Nhat, and N. D. Q. Bui. Swe-evo: Benchmarking coding agents in long-horizon software evolution scenarios, 2025. URL https://arxiv.org/abs/2512.18470.

The Terminal-Bench Team. Terminal-bench: A benchmark for AI agents in terminal environments, Apr 2025. URL https://github.com/laude-institute/terminal-bench.

R. Thomas. Breaking the spell of vibe coding. https://www.fast.ai/posts/2026-01-28-dark-flow/, 2026.

J. Yang, C. E. Jimenez, A. Wettig, et al. SWE-agent: Agent-computer interfaces enable automated software engineering, 2024. URL https://arxiv.org/abs/2405.15793.

S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao. ReAct: Synergizing reasoning and acting in language models, 2022. URL https://arxiv.org/abs/2210.03629.

A. SWE-AGI Task Suite

Table 8 | SWE-AGI task suite (22 tasks, 7 categories). Core LOC (excluding tests and tooling) is reported as a coarse magnitude estimate (derived from benchmarked agent implementations and excluding public and private tests). Rows are sorted by core LOC within each category.

Task (`id`: title)	Difficulty	Core LOC	Key complexity drivers
Totals: 22 tasks (Easy=6, Medium=8, Hard=8).			
Template and Domain-Specific Languages			
`pug`: Pug Template Language	Medium	$\sim 5 \times 10^3$	Indentation semantics, mixins/blocks, scope/inclusion, error localization
`jq`: JQ Query Language Interpreter	Hard	$\sim 7 \times 10^3$	Lexer/parser, stream semantics (0..N outputs), built-ins, error modes
Data Serialization and Configuration Formats			
`csv`: CSV Parser (RFC 4180)	Easy	$\sim 10^3$	Quoting/escaping, multiline fields, line ending edge cases, invalid patterns
`ini`: INI Parser	Easy	$\sim 10^3$	Section/key parsing, escaping rules, normalization, error handling
`yaml`: YAML 1.2 Parser	Medium	$\sim 3 \times 10^3$	Indentation/block structure, anchors/tags, scalars, error recovery
`toml`: TOML 1.0 Parser	Medium	$\sim 3 \times 10^3$	Dotted keys, array-of-tables, datetime/float rules, UTF-8 + diagnostics
Markup and Document Formats			
`xml`: XML 1.0 + Namespaces	Medium	$\sim 3 \times 10^3$	Well-formedness, namespaces, entities/DTD subset, error handling, streaming/DOM tradeoffs
`html5`: HTML5 Parser	Hard	$\sim 10^4$	Tokenization + tree builder state machines, error recovery, entities, broad conformance
Programming Language Front-Ends			
`c99`: C99 Parser	Hard	$\sim 5 \times 10^3$	Declarators/type system, precedence/ambiguity, AST + symbols, error recovery
`lua`: Lua 5.4 Interpreter	Hard	$\sim 5 \times 10^3$	VM/bytecode, tables + metatables, closures, coroutines, GC scope
`ecma262`: ECMAScript Interpreter (ECMA-262 subset)	Hard	$\sim 7 \times 10^3$	Parsing + semantics, runtime objects, corner cases exercised by suite
`python`: Python Interpreter (subset)	Hard	$\sim 7 \times 10^3$	Indentation lexing, object model, exceptions, scoping/closures, built-ins
`r6rs`: R6RS Scheme Interpreter (subset)	Hard	$\sim 7 \times 10^3$	Reader, macro system, evaluator/runtime, exact printing semantics
Binary Formats and Streaming Decoders			
`git_object`: Git Object Parser (loose objects)	Easy	$\sim 10^3$	zlib integration, header parsing, hashing, boundary/error handling
`protobuf`: Protocol Buffers (streaming codec)	Easy	$\sim 10^3$	Varint/zigzag, length-delimited fields, chunked reads, malformed input handling
`zip`: ZIP File Parser	Medium	$\sim 3 \times 10^3$	Central directory, Zip64, streaming reads, CRC/validation, encoding details
`capnp`: Cap'n Proto Binary Format	Medium	$\sim 3 \times 10^3$	Packed encoding, pointers/segments, far pointers, boundary safety
`wasm`: WASM Decoder + Validator	Medium	$\sim 5 \times 10^3$	LEB128, section/index consistency, validation rules, precise error behavior
Networking and Protocol State Machines			
`uri`: URI Parser (RFC 3986)	Easy	$\sim 10^3$	Normalization and resolution rules, encoding constraints, error behavior
`hpack`: HPACK Decoder/Encoder (RFC 7541)	Easy	$\sim 10^3$	Huffman coding, dynamic table management, header field semantics
`url`: URL Parser (WHATWG)	Medium	$\sim 3 \times 10^3$	Canonicalization, relative resolution, percent-encoding, IDNA/Punycode scope
Automated Reasoning and SAT Solving			
`cdcl`: CDCL SAT Solver	Hard	$\sim 2 \times 10^3$	Unit propagation, clause learning, backtracking/heuristics, data-structure efficiency

B. Detailed Results on SWE Behaviors

Table 9 collects the per-task behavior stats tables referenced in Section 3.3: the first part reports gpt-5.3-codex and gpt-5.2-codex, and the continuation reports claude-opus-4.6 and claude-opus-4.5. Percentages denote the share of logged tool actions assigned to each behavior category (Spec, Plan, Read, Write, Debug, Hyg, Ext, Other); for readability, the top-3 behavior shares per row are bold. In Table 9, the *Action* column reports the counted logged actions for that task run and should be interpreted as a coarse proxy for interaction volume rather than a normalized efficiency measure, since logging granularity varies across agent front-ends and runs.

Table 9 | Per-task behavior stats for gpt-5.3-codex and gpt-5.2-codex (percent of logged actions). Top-3 behavior shares per row are bold.

Task	Model	Action	Spec	Plan	Read	Write	Debug	Hyg	Ext	Other
pug	gpt-5.3-codex	708	5.5%	0.0%	**35.6%**	**18.4%**	**15.0%**	1.0%	13.6%	11.0%
	gpt-5.2-codex	5093	5.4%	0.0%	**54.0%**	**19.2%**	**10.3%**	1.3%	2.7%	7.0%
jq	gpt-5.3-codex	435	2.8%	0.0%	**32.0%**	20.7%	**22.5%**	0.2%	0.0%	**21.8%**
	gpt-5.2-codex	522	3.3%	0.0%	**45.2%**	**31.6%**	**16.7%**	3.3%	0.0%	0.0%
csv	gpt-5.3-codex	84	8.3%	0.0%	**48.8%**	**22.6%**	**17.9%**	0.0%	0.0%	2.4%
	gpt-5.2-codex	143	7.7%	0.7%	**56.6%**	**19.6%**	**9.1%**	5.6%	0.0%	0.7%
ini	gpt-5.3-codex	99	5.1%	0.0%	**40.4%**	**29.3%**	**25.3%**	0.0%	0.0%	0.0%
	gpt-5.2-codex	176	4.0%	0.0%	**42.6%**	**31.8%**	**15.9%**	5.1%	0.0%	0.6%
yaml	gpt-5.3-codex	288	6.6%	0.0%	**50.0%**	**19.8%**	**17.7%**	0.3%	0.7%	4.9%
	gpt-5.2-codex	721	3.5%	0.1%	**40.9%**	**34.0%**	**16.0%**	3.6%	0.8%	1.1%
toml	gpt-5.3-codex	237	12.2%	0.0%	**33.3%**	**14.3%**	**30.4%**	3.4%	5.5%	0.8%
	gpt-5.2-codex	474	8.9%	0.0%	**39.0%**	**28.1%**	**19.2%**	4.0%	0.0%	0.8%
xml	gpt-5.3-codex	339	5.9%	0.0%	**30.1%**	**32.2%**	**22.4%**	0.0%	8.0%	1.5%
	gpt-5.2-codex	483	4.1%	0.2%	**42.2%**	**32.5%**	**16.4%**	4.6%	0.0%	0.0%
html5	gpt-5.3-codex	329	7.9%	0.0%	**30.1%**	**20.1%**	**29.5%**	0.6%	7.3%	4.6%
	gpt-5.2-codex	386	10.1%	0.3%	**33.9%**	**28.2%**	**21.0%**	0.5%	0.0%	6.0%
c99	gpt-5.3-codex	218	6.9%	0.0%	**39.9%**	**25.2%**	**19.7%**	0.5%	0.5%	7.3%
	gpt-5.2-codex	580	10.9%	0.2%	**48.6%**	**25.7%**	**11.9%**	2.4%	0.0%	0.3%
lua	gpt-5.3-codex	215	4.2%	0.0%	**42.3%**	**35.8%**	**16.7%**	0.0%	0.0%	0.9%
	gpt-5.2-codex	568	1.8%	0.2%	**52.6%**	**34.7%**	**9.3%**	1.1%	0.0%	0.4%
ecma262	gpt-5.3-codex	201	10.0%	0.0%	**48.3%**	**21.9%**	**14.9%**	0.5%	2.5%	2.0%
	gpt-5.2-codex	9794	0.8%	0.0%	**71.2%**	**17.3%**	**7.8%**	0.4%	0.3%	2.1%
python	gpt-5.3-codex	419	5.3%	0.0%	**56.1%**	**20.3%**	**12.6%**	0.2%	2.9%	2.6%
	gpt-5.2-codex	511	3.1%	0.0%	**55.4%**	**24.9%**	**10.2%**	1.6%	0.0%	4.9%
r6rs	gpt-5.3-codex	377	8.2%	0.0%	**46.2%**	12.5%	**15.9%**	1.6%	1.6%	**14.1%**
	gpt-5.2-codex	654	**10.4%**	0.2%	**49.1%**	**28.9%**	10.2%	0.2%	0.0%	1.1%
git_object	gpt-5.3-codex	125	4.0%	0.0%	**59.2%**	**24.8%**	**9.6%**	0.8%	0.0%	1.6%
	gpt-5.2-codex	324	4.6%	0.3%	**54.3%**	**23.5%**	**12.7%**	3.4%	0.0%	1.2%
protobuf	gpt-5.3-codex	109	**13.8%**	0.0%	**46.8%**	**24.8%**	12.8%	1.8%	0.0%	0.0%
	gpt-5.2-codex	136	**12.5%**	0.7%	**54.4%**	**20.6%**	8.1%	2.2%	0.0%	1.5%
zip	gpt-5.3-codex	128	5.5%	0.0%	**52.3%**	**22.7%**	**12.5%**	0.0%	0.0%	7.0%
	gpt-5.2-codex	693	2.5%	0.0%	**42.6%**	**12.0%**	7.8%	1.2%	1.9%	**32.2%**
capnp	gpt-5.3-codex	194	13.4%	0.0%	**42.8%**	**19.1%**	**21.6%**	1.0%	0.0%	2.1%
	gpt-5.2-codex	265	**12.5%**	0.0%	**55.8%**	**18.1%**	10.9%	1.9%	0.0%	0.8%
wasm	gpt-5.3-codex	263	10.3%	0.0%	**42.6%**	**14.8%**	**16.3%**	0.0%	4.2%	11.8%
	gpt-5.2-codex	491	10.4%	0.2%	**49.1%**	**21.0%**	**13.4%**	2.4%	3.3%	0.2%
uri	gpt-5.3-codex	95	12.6%	0.0%	**33.7%**	**25.3%**	**28.4%**	0.0%	0.0%	0.0%
	gpt-5.2-codex	187	4.3%	0.5%	**49.2%**	**28.9%**	**14.4%**	2.1%	0.0%	0.5%
hpack	gpt-5.3-codex	142	12.0%	0.7%	**44.4%**	**22.5%**	**16.2%**	0.0%	0.0%	4.2%
	gpt-5.2-codex	204	**10.8%**	0.5%	**43.1%**	**26.5%**	10.3%	2.5%	0.0%	6.4%
url	gpt-5.3-codex	125	7.2%	0.0%	**53.6%**	**16.0%**	**17.6%**	0.8%	0.0%	4.8%
	gpt-5.2-codex	338	**17.8%**	0.0%	**48.5%**	**16.6%**	10.4%	0.6%	0.0%	6.2%
cdcl	gpt-5.3-codex	216	4.2%	0.0%	**35.2%**	**18.5%**	**28.2%**	0.5%	0.0%	13.4%
	gpt-5.2-codex	393	2.3%	0.3%	**35.9%**	**31.3%**	**14.2%**	13.5%	0.0%	2.5%

Table 9 | Per-task behavior stats (continued) for claude-opus-4.6 and claude-opus-4.5 (percent of logged actions). Top-3 behavior shares per row are bold.

Task	Model	Action	Spec	Plan	Read	Write	Debug	Hyg	Ext	Other
pug	claude-opus-4.6	528	6.1%	8.3%	**50.8%**	**16.1%**	**12.9%**	0.2%	0.9%	4.7%
	claude-opus-4.5	309	5.8%	3.2%	**27.8%**	**35.3%**	**20.1%**	0.0%	0.6%	7.1%
jq	claude-opus-4.6	459	3.7%	6.5%	**52.5%**	**22.0%**	**11.5%**	0.0%	2.6%	1.1%
	claude-opus-4.5	464	3.4%	3.0%	**47.4%**	**27.4%**	**18.1%**	0.2%	0.0%	0.4%
csv	claude-opus-4.6	214	7.9%	5.6%	**30.4%**	**27.6%**	**26.2%**	0.5%	0.9%	0.9%
	claude-opus-4.5	149	4.7%	6.7%	**18.1%**	**32.9%**	**34.2%**	0.0%	2.0%	1.3%
ini	claude-opus-4.6	146	9.6%	6.8%	**34.9%**	**24.0%**	**21.9%**	0.0%	1.4%	1.4%
	claude-opus-4.5	97	11.3%	6.2%	**21.6%**	**30.9%**	**29.9%**	0.0%	0.0%	0.0%
yaml	claude-opus-4.6	513	6.0%	4.5%	**47.8%**	**11.5%**	**20.9%**	0.2%	3.3%	5.8%
	claude-opus-4.5	491	3.9%	3.1%	**42.4%**	**23.4%**	**24.4%**	0.0%	0.8%	2.0%
toml	claude-opus-4.6	3093	8.6%	5.3%	**36.6%**	14.2%	**15.3%**	0.2%	**18.5%**	1.3%
	claude-opus-4.5	292	9.2%	3.8%	**34.2%**	**26.7%**	**24.0%**	1.4%	0.0%	0.7%
xml	claude-opus-4.6	542	6.5%	4.4%	**39.5%**	**28.0%**	**17.3%**	0.0%	4.1%	0.2%
	claude-opus-4.5	534	2.6%	6.4%	**32.6%**	**26.4%**	**31.8%**	0.0%	0.0%	0.2%
html5	claude-opus-4.6	2179	5.8%	4.4%	**48.5%**	10.3%	**12.1%**	0.0%	**14.0%**	4.9%
	claude-opus-4.5	181	16.0%	5.5%	**16.6%**	**36.5%**	**23.2%**	0.0%	0.0%	2.2%
c99	claude-opus-4.6	427	**13.8%**	4.7%	**39.1%**	**19.9%**	11.0%	0.0%	0.0%	11.5%
	claude-opus-4.5	290	11.0%	5.2%	**34.8%**	**20.7%**	**25.9%**	0.7%	0.0%	1.7%
lua	claude-opus-4.6	425	8.0%	7.5%	**38.4%**	**21.2%**	**20.5%**	0.2%	0.5%	3.8%
	claude-opus-4.5	465	2.8%	5.8%	**51.4%**	**17.4%**	**18.9%**	0.0%	0.0%	3.7%
ecma262	claude-opus-4.6	2384	6.5%	**10.6%**	**53.4%**	10.2%	**16.9%**	0.0%	1.3%	1.1%
	claude-opus-4.5	347	6.1%	3.5%	**42.1%**	**30.3%**	**13.0%**	0.0%	0.0%	5.2%
python	claude-opus-4.6	2190	5.1%	6.0%	**57.2%**	**12.1%**	**14.4%**	0.1%	1.4%	3.7%
	claude-opus-4.5	848	3.2%	2.4%	**50.1%**	**23.5%**	**18.8%**	0.0%	0.9%	1.2%
r6rs	claude-opus-4.6	3146	7.5%	5.8%	**48.4%**	**14.8%**	**18.9%**	0.0%	0.4%	4.1%
	claude-opus-4.5	539	6.5%	4.6%	**44.5%**	**22.1%**	**21.3%**	0.0%	0.0%	0.9%
git_object	claude-opus-4.6	206	**12.6%**	5.3%	**52.4%**	**16.5%**	10.2%	0.5%	0.0%	2.4%
	claude-opus-4.5	261	10.0%	3.4%	**33.7%**	**24.1%**	**24.1%**	0.0%	0.0%	4.6%
protobuf	claude-opus-4.6	97	14.4%	8.2%	**27.8%**	**32.0%**	**17.5%**	0.0%	0.0%	0.0%
	claude-opus-4.5	94	**26.6%**	16.0%	**20.2%**	**26.6%**	10.6%	0.0%	0.0%	0.0%
zip	claude-opus-4.6	1990	3.6%	6.5%	**57.3%**	7.1%	**10.2%**	0.0%	4.9%	**10.5%**
	claude-opus-4.5	595	4.9%	1.2%	**44.2%**	**21.3%**	**19.7%**	0.2%	0.0%	8.6%
capnp	claude-opus-4.6	195	**16.4%**	5.6%	**39.0%**	**26.2%**	11.8%	0.5%	0.0%	0.5%
	claude-opus-4.5	87	**21.8%**	8.0%	**36.8%**	**20.7%**	11.5%	1.1%	0.0%	0.0%
wasm	claude-opus-4.6	211	**22.7%**	5.2%	**29.4%**	**19.9%**	17.1%	0.0%	0.5%	5.2%
	claude-opus-4.5	450	6.7%	2.2%	**32.2%**	**24.0%**	**34.0%**	0.0%	0.0%	0.9%
uri	claude-opus-4.6	101	**19.8%**	5.9%	**27.7%**	**24.8%**	18.8%	1.0%	0.0%	2.0%
	claude-opus-4.5	124	**22.6%**	6.5%	**22.6%**	**25.0%**	21.8%	0.8%	0.0%	0.8%
hpack	claude-opus-4.6	136	11.8%	8.8%	**27.2%**	**24.3%**	8.8%	0.0%	5.1%	**14.0%**
	claude-opus-4.5	143	**22.4%**	6.3%	**25.9%**	**25.2%**	19.6%	0.0%	0.0%	0.7%
url	claude-opus-4.6	432	**22.7%**	4.4%	**22.0%**	**16.0%**	14.6%	0.2%	12.7%	7.4%
	claude-opus-4.5	290	11.0%	5.9%	**29.7%**	**31.4%**	**20.0%**	0.0%	0.0%	2.1%
cdcl	claude-opus-4.6	772	6.0%	7.5%	**44.3%**	**15.9%**	**22.4%**	0.1%	2.3%	1.4%
	claude-opus-4.5	334	2.7%	8.4%	**32.6%**	**27.2%**	**28.7%**	0.0%	0.0%	0.3%