CAN AGENTS DESIGN LIBRARIES FOR AGENTS?

Anonymous authors
Paper under double-blind review

ABSTRACT

Agents increasingly build on code written by other agents, and they reimplement rather than reuse, growing the codebases later agents must work in. To measure how well agents design libraries for other agents, we introduce LibraryDesignBench, a two-phase benchmark in which an agent implements a full-featured library from a specification that defines required capabilities and potential use cases without prescribing the design. We evaluate the library through the correctness and simplicity of programs written by three user agents from different model families. Across 242 expert-validated programming problems spanning 15 library-design tasks in four languages, agent designers reproduce the abstractions of their human-authored counterparts on eleven of fifteen tasks, and downstream agents adopt both kinds of library but underuse them, reimplementing capabilities the library already provides. Our audit identifies rigid or hard-to-use interfaces as the leading obstacles in the sampled excess-code failures. We experiment with a targeted intervention: giving designers more prescriptive agent-first guidance and having them test their library with subagents improves downstream scores and yields simpler programs. LibraryDesignBench provides both a testbed for evaluating library-design practices for agent users and an initial design baseline that improves downstream reuse.

1 INTRODUCTION

Software engineering as a discipline would not exist without skilled engineers designing libraries, frameworks, SDKs, etc., with *opinionated design decisions* that make future work easier. Yet, we are fast approaching an inflection point where agents will work more with code designed by another agent than by a human engineer. It raises a fundamental question: *can agents design libraries that other agents can leverage?* Poorly designed libraries will hamper future agents' performance in both correctness and code volume – requiring more human effort and intervention to repair. Answering it demands evaluating the library through downstream agent use, not correctness tests alone.

The core roadblock is that grading a generated library is nontrivial. Tests only check that the library is correct, not whether it helps the agents that use it. Grading the interface against a specification requires signatures or constructs, which prescribe the very design we want to measure (Ding et al., 2026; Zhao et al., 2025; Liu et al., 2025; Peng et al., 2026; Gautam et al., 2025). Agentic validation (Ehrenberg et al., 2026) and static metrics score the implementation, not its usability. Human review measures what humans prefer, not what agents need. *The only faithful way to evaluate a library written for agents is to observe agents use it.*

We therefore propose **LibraryDesignBench**, a two-phase evaluation that assesses an agent's ability to design a library by measuring how downstream agents use it to write better programs. The benchmark comprises fifteen library-design problems and 242 expert-validated downstream programming problems across four languages. In the **Design Phase**, the agent under evaluation implements a full-featured library from a specification that defines required capabilities while leaving interfaces and abstractions open. In the **Evaluation Phase**, multiple different, less capable user agents use that library to implement downstream programs. We evaluate the library by the correctness and simplicity of these programs, using reference solutions built with real production libraries. We define simplicity using capped reference-to-program ratios averaged over four static size and complexity measures.

Opus 5.5 scores highest (48.9), 2.3 points above the production library. In eleven of fifteen tasks, designers reproduce production-library abstractions. Our audit classifies 64% of sampled excess-code cases under rigid or hard-to-use interfaces, and only 14% under missing capabilities.

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

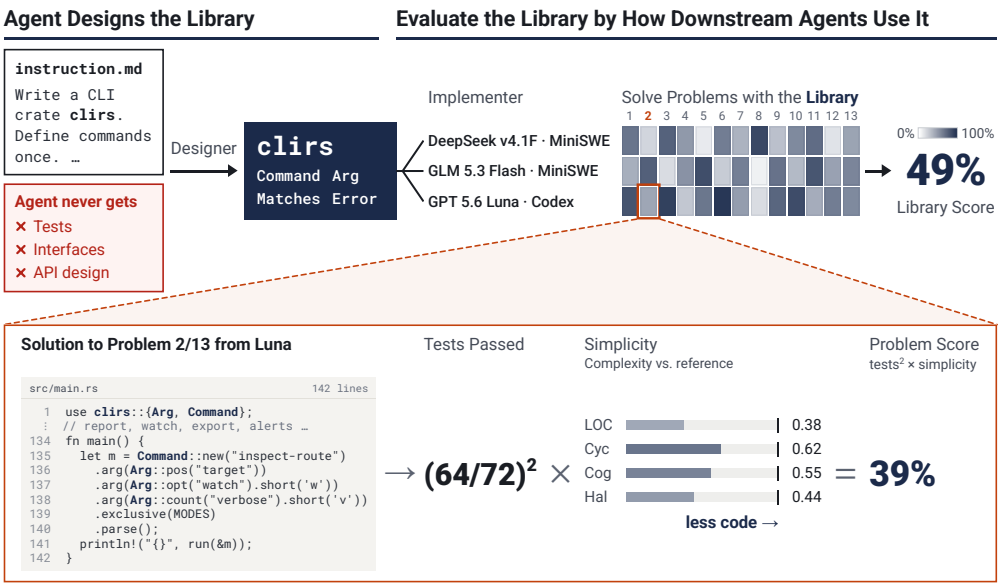


Figure 1: **LibraryDesignBench's two-phase setup evaluates the library through real usage.** The agent under evaluation designs the library from a non-prescriptive specification. Three downstream agents then solve problems with it; the score reflects how correct and simple their programs are.

A library's score also depends on how well implementers use it. Even with the production library, implementers reach a simplicity of only 61.5, so part of the gap to the reference comes from the implementer, not the design. To measure this separately, we fix the library to the production library and evaluate 8 models as implementers, which we call **LibraryUseBench**. Opus 5.5 scores best at 66.9. For GPT-5.6 Luna, higher reasoning effort mainly improves correctness, a more prescriptive prompt mainly improves library use, and its solutions remain far longer than the reference.

Because agents currently reproduce production-library abstractions, we next test whether more prescriptive agent-first guidance helps. We have GPT-6 Astra sketch consumer programs first, ship them as runnable usage examples, and test its design with subagents. This raises the score by 2.3 points, mainly from 6.8% higher simplicity, yet it still falls just short of the production library. *Designing libraries for agents differs from designing them for humans, and remains an open problem.*

Our contributions are:

- **LibraryDesignBench.** We introduce LibraryDesignBench which evaluates agent-designed libraries only by observing how well downstream agents can leverage them. (Section 2)
- **How agents design and use libraries, and why they fail.** Agent-designed libraries reproduce production-library abstractions (Section 3.1), and our audit attributes most excess-code failures to rigid or hard-to-use interfaces, not missing capabilities (Section 3.2). Even with a production library, agents exploit it only when pushed and still write far more code than an expert (Section 3.3).
- **Prompting Interventions.** More prescriptive agent-first guidance, combining consumer-first API sketches, runnable usage examples, and testing with subagents, reduces exported-name overlap with production libraries, improves downstream scores, and yields simpler programs (Section 3.4).[1]

2 LIBRARYDESIGNBENCH

The core goal of LibraryDesignBench is to measure the quality of a library designed by an agent *only* by observing how downstream agents utilize it. A single task consists of two phases:

[1]Code and data: https://anonymous.4open.science/r/librarydesignbench-D113

- **Design Phase**: tasks the *agent under evaluation*, π_θ, with building the library L.
- **Evaluation Phase**: downstream implementers, π_u, solve tasks using L.

To score L, we consider both *correctness* through the task's test suite and *simplicity* compared to reference solutions written idiomatically with a real production library.

Running problem: CLI crate for Rust

Design Phase: Design `clirs`, a Rust CLI-parsing crate. The spec names commands, flags, options, typed values, arity, cross-argument relations, and generated help. The environment is offline with no argument-parsing library, so the agent builds the parser from scratch.
Evaluation Phase: A fresh agent builds one of 13 CLI tools with the authored crate. `site-plan`, for instance, needs two subcommands sharing a group of lifetime options of which exactly one must be given, a repeatable `--resource SRC DEST` with fixed two-value arity, four equivalent option spellings (`-ndocs`, `-n=docs`, `--site-name=docs`, `-n docs`), and grouped 72-column help. Other tools stress nested command trees with aliases, inherited globals, reusable argument groups, and parser-consistent introspection.
Production library: clap. Behavioral tests establish correctness; program size relative to the reference clap solution measures library value.

2.1 Evaluating Libraries Through Real Observation

LibraryDesignBench measures how well an agent can create a library, L, that helps the future agents who use it. Direct test suites, or even agentic verifiers, can only measure *whether this library is correct*. They cannot measure how it will impact agents trying to use it to solve *real* tasks.

Design Phase. The agent under evaluation, π_θ, implements L given an instruction, I. The instruction leaves interfaces and abstractions open, forcing π_θ to reason through which abstractions are needed and which are not. It is provided a list of functionality it must support and two to three example usages drawn from the task's own evaluation problems. These "visible" problems give the agent the grounding it needs for understanding how its library will be used, similar to how real engineers express a library spec. The visible problems remain in the scored evaluation set, analogous to visible test cases. Beyond the named capabilities, the general instructions require functionality users would reasonably expect of the library (Appendix D). We also instruct agents that the primary users of L will be coding agents and that senior engineers will review their work. Finally, agents must package their library with a language-specific package manager so that it installs.

Evaluation Phase. Each task contains a set of problems, $\mathcal{P}$, that each implementer $\pi_u \in U$ will solve using L. Problems must be solvable with or without a library. Implementers are explicitly instructed to make their solutions "thin adapters" over the libraries and to ensure they read the documentation (Listing 1). We want to elicit the library's ability to be exploited to write as little code as possible, not the implementer's ability to recognize when a library helps.

2.2 Scoring the Quality of a Library

A library is only valuable if the underlying implementations are *correct* and it enables writing *simpler* programs. A library whose programs are shorter but incorrect, or correct but no shorter than without it, provides little value. Thus, we *score* a L according to the product of correctness and simplicity:

$$\text{score}(L) = \frac{1}{|\mathcal{P}| \cdot |U|} \sum_{u \in U} \sum_{x_i \in \mathcal{P}} q_i(y_i)^2 \cdot \rho_i(y_i), \qquad y_i \sim \pi_u(x_i \mid L). \tag{1}$$

Here $q_i(y_i) \in [0, 1]$ is the fraction of tests passed and $\rho_i(y_i)$ measures static simplicity relative to the problem's reference solution. Each implementer produces one solution per problem, and we report scores multiplied by 100. We square the test-pass fraction to prioritize correctness while retaining graded credit for partially correct programs, so passing 80% of tests at the reference's size (0.64) scores below passing every test at $1.5\times$ its size (0.67). If L cannot be installed, its score is always 0 across all problems. Appendix A derives the aggregation, standard errors, and confidence intervals.

Simplicity. A library should reduce the code needed to solve a task, first and foremost. Let y_i^* be the fixed optimized reference solution for problem i. We define

$$\rho_i(y_i) = \frac{1}{|M|} \sum_{m \in M} \min\left\{ \frac{m(y_i^*)}{m(y_i)}, 1 \right\}. \tag{2}$$

We utilize a set of static metrics, M, which reduces dependence on any single static metric (e.g., a parser that accepts every option spelling removes branches, not just lines). Averaging over M keeps ρ_i on the same scale as a single metric ratio, so a solution that matches its reference on every metric scores 1 and one twice its size on every metric scores 0.5. Capping each ratio at one bounds the contribution of programs smaller than the reference. If a metric is zero for the solution, its ratio is set to 1. If no solution is produced, or the solution has syntax errors, its simplicity is zero.

M consists of static counts that quantify residual code without grading conformity to an interface:

$$M = \{\text{Cyclomatic Complexity}, \text{Cognitive Complexity}, \text{Halstead Volume}, \text{Source Lines of Code}\},$$

with definitions and language-specific counting rules detailed in Appendix B for Cyclomatic Complexity (McCabe, 1976), Cognitive Complexity (Campbell, 2018), and Halstead Volume (Halstead, 1977). We measure source lines of code after applying a language-standard formatter, excluding comments and blank lines. Metrics are computed over eligible source files using the counting and aggregation rules in Appendix B. Cognitive complexity targets human comprehension, but it captures a different kind of complexity than cyclomatic, and agents spend more tokens and revisit more files on code that scores high on it and violates more static-analysis rules (Trivedi & Schmitt, 2026).

Reference solutions. The reference y_i^* is a fixed program per problem written using the mature production library $L^\star$ and optimized for idiomatic use of its abstractions, rather than code golfing. Library experts developed and optimized the reference programs with agent assistance. We apply the same formatting and measurement procedures to reference and generated programs. For example, the `clirs` references use `clap`'s declarative `derive` pattern, not its builder syntax.

Implementer configurations. Each $u \in U$ identifies a complete implementer configuration, including its model, harness, and inference settings. We use the same fixed set across all generated libraries and comparison conditions. Equation 1 intentionally weights every implementer equally.

2.3 Benchmark Construction

We now detail the construction of LibraryDesignBench, which yielded fifteen tasks across four programming languages. We selected libraries to use as tasks based on their age, complexity, and the number of interface decisions a designer must make. The Evaluation Phase problem desiderata are:

1. **Realistic task.** A problem reflects a realistic use case for the library.
2. **Library Headroom.** A problem is valuable to LibraryDesignBench if the library reduces significant amounts of code through composition and interaction of features.
3. **Solvable Without a Library.** A problem written such that only one library could reasonably solve it is not a fair problem for LibraryDesignBench. Thus, every problem must be solvable without any library available.

Each problem is built through a multi-stage agentic pipeline that isolates the library's functionality, seeded by real usages of the library from permissively licensed repositories. First, an extraction agent reduces the seed to a minimal program that exercises the library, removing application-specific logic. Next, a rewriting agent produces a library-free equivalent and a test suite on which both implementations must agree. We then verify that every test is solvable from the task instructions and workspace alone. Finally, library experts review each problem, rewrite its instructions, and strengthen its tests against solutions that omit required behavior; a second expert audits each review.

3 Evaluating Frontier Models on LibraryDesignBench

Each designer runs in mini-SWE-agent unless noted, with $K = 3$ libraries per task and 3 implementers per library (2,178 evaluated problems per designer). We answer four research questions:

Table 1: **Overall results for each library setup by designer.** All results average the same implementer set. Designers use mini-SWE-agent (Yang et al., 2025) unless another harness is in parentheses (CC is Claude Code). Production and No library are settings in which the implementer is given a human-written library or no library at all (Listing 2), respectively. "Library $" is the average cost, in USD, to generate a single library, while "Problem $" is the average implementer cost per evaluation problem. "% Pass" is the mean share of tests passed, not of fully solved problems. Scores show 95% CIs; other values show clustered standard errors (Appendix A). **Bold** marks the best per column.

Model (Harness)	Score (↑)	% Pass (↑)	Simplicity (↑)	Library $ (↓)	Problem $ (↓)
DeepSeek V4 Pro	31.2 [30.4, 32.1]	84.9 ±1.7	42.0 ±1.1	$0.31 ±0.02	$0.175 ±0.008
Fable 5.1	47.5 [46.4, 48.5]	86.1 ±1.6	62.7 ±1.8	$14.81 ±1.37	$0.198 ±0.013
Fable 5.1 (CC)	39.9 [37.4, 42.4]	84.9 ±1.6	58.3 ±2.5	$14.39 ±2.23	$0.211 ±0.014
GLM 5.3	41.8 [40.8, 42.9]	84.1 ±1.7	57.1 ±1.7	$21.06 ±1.89	$0.234 ±0.013
GPT-5.6 Sol (Codex)	39.5 [38.4, 40.5]	84.1 ±2.1	52.9 ±1.2	$2.14 ±0.20	$0.199 ±0.011
GPT-6 Astra	45.1 [44.3, 45.8]	85.7 ±1.7	58.7 ±1.5	$3.63 ±0.24	$0.155 ±0.008
GPT-6 Astra (Codex)	44.1 [43.4, 44.9]	85.4 ±1.9	58.5 ±1.6	$4.51 ±0.40	$0.194 ±0.012
GPT-6 Sol	38.5 [37.8, 39.2]	85.2 ±1.9	51.6 ±1.3	**$0.30** ±0.02	$0.150 ±0.007
Grok 4.6	39.7 [38.6, 40.7]	84.6 ±1.6	54.1 ±1.6	$2.63 ±0.23	$0.225 ±0.012
Kimi K3	44.0 [43.0, 45.0]	86.0 ±1.5	58.6 ±1.7	$8.70 ±1.47	$0.176 ±0.010
Opus 5.5	**48.9** [48.0, 49.9]	**86.6** ±1.4	**64.5** ±1.8	$9.62 ±1.03	$0.174 ±0.011
Production Library	46.6 [46.1, 47.2]	85.4 ±1.0	61.5 ±0.9	–	$0.240 ±0.020
No Library	34.4 [33.9, 34.9]	86.4 ±1.0	46.3 ±1.1	–	**$0.098** ±0.006

RQ1. Can agents design libraries that improve other agents? Yes. The strongest designer exceeds the production-library baseline by 2.3 points (4.9% relative), while designers reproduce production-library abstractions on eleven of fifteen tasks. (Section 3.1)

RQ2. Why do agents struggle with agent-designed libraries? In sampled partially passing solutions, our audit classifies 64% of excess-code cases under rigid or hard-to-use interfaces, not missing capabilities. (Section 3.2)

RQ3. How can implementers better leverage libraries? More prescriptive prompts and higher reasoning effort raise the score by 23% and 63% relative; prescription drives library use, effort drives correctness, and solutions stay far longer than the reference. (Section 3.3)

RQ4. Can agentic design patterns improve implementer performance? Yes, modestly. More prescriptive agent-first guidance raises the score by 2.3 points. (Section 3.4)

Setup. Agents have no internet access, 4 hours to design, and 1 hour and $2.50 per problem; unfinished solutions are scored as is (1.7% of trials; Appendix K). Details are in Appendix C.

3.1 Agent Design Quality

Table 1 highlights our overall results. Downstream correctness does not separate designers, as every setup passes 84.1%–86.6% of tests. The no-library condition reaches 86.4%, within 0.2 percentage points of the highest mean test-pass rate. Score differences primarily reflect how simple downstream agents' solutions are. Production libraries add 12.3 ± 0.4 points over no library at $\$0.14 \pm \$0.02$ more per problem. On the other end, DeepSeek V4 Pro's library scores 9.2% below no library. Harm is most common in Haskell, where agent-written libraries score below no library in about 70% of the 33 (designer, Haskell task) pairs. On the 3 tasks where no library beats production, Opus 5.5's libraries trail no library by 3.2 points (Figure 6). The harness also matters: Fable 5.1 scores 47.5 in mini-SWE-agent but 39.9 in Claude Code. We therefore report harness variants separately.

Agents reproduce production-library abstractions. Designers converge on the same design in eleven of fifteen tasks. We assessed this by inspecting the libraries, with agent verification. In `clirs`, Astra and Fable copy `clap`'s builder design, not its shorter derive macro (Figure 2).

Table 2: **Results per implementer across library setups.** Only mini-SWE-agent designers are shown. "Problem tokens" is the mean number of tokens per problem.

Setup	Score (95% CI, ↑)	% Pass (↑)	Simplicity (↑)	Problem tokens (↓)
Implementer: **DeepSeek V4.1 Flash**				
Astra	46.9 [45.8, 47.9]	87.4 ±1.7	58.5 ±1.4	5.80 ±0.34M
GPT-6 Sol	41.3 [40.2, 42.5]	87.0 ±1.8	51.6 ±1.2	4.86 ±0.28M
Fable	50.2 [48.9, 51.5]	87.8 ±1.7	63.2 ±1.8	6.35 ±0.35M
Opus 5.5	**51.6** [50.4, 52.8]	88.4 ±1.5	**64.3** ±1.8	6.47 ±0.36M
GLM 5.3	45.1 [43.8, 46.4]	86.2 ±1.8	57.3 ±1.6	8.43 ±0.46M
Grok	42.1 [40.7, 43.4]	86.6 ±1.7	53.8 ±1.4	7.07 ±0.42M
DeepSeek V4 Pro	33.3 [32.4, 34.3]	87.5 ±1.8	41.3 ±1.3	5.92 ±0.29M
Kimi K3	46.4 [45.2, 47.7]	87.8 ±1.5	57.9 ±1.6	6.23 ±0.35M
Production library	49.3 [48.5, 50.2]	86.2 ±3.8	62.0 ±2.4	6.49 ±0.97M
No library	37.0 [36.0, 38.0]	**89.1** ±2.9	46.0 ±3.9	**3.03** ±0.39M
Implementer: **GPT-5.6 Luna**				
Astra	44.3 [43.4, 45.2]	85.3 ±1.8	57.3 ±1.7	3.00 ±0.22M
GPT-6 Sol	36.8 [35.9, 37.7]	83.4 ±2.1	49.3 ±1.5	2.24 ±0.19M
Fable	46.2 [45.3, 47.2]	85.6 ±1.6	60.8 ±1.9	3.64 ±0.27M
Opus 5.5	**47.3** [46.3, 48.3]	85.9 ±1.5	**62.1** ±2.0	3.35 ±0.24M
GLM 5.3	40.5 [39.5, 41.5]	83.5 ±1.8	55.3 ±1.7	3.93 ±0.26M
Grok	39.0 [37.7, 40.4]	83.1 ±1.9	53.1 ±1.7	3.22 ±0.21M
DeepSeek V4 Pro	31.2 [30.1, 32.4]	83.1 ±1.8	42.6 ±1.2	2.79 ±0.18M
Kimi K3	42.6 [41.2, 43.9]	84.7 ±1.6	57.1 ±1.8	2.99 ±0.20M
Production library	45.5 [44.6, 46.3]	84.5 ±2.7	60.4 ±2.8	4.26 ±0.73M
No library	33.3 [32.6, 33.9]	**86.1** ±3.0	44.3 ±2.6	**1.46** ±0.31M
Implementer: **GLM 5.3 Flash**				
Astra	44.1 [42.7, 45.4]	84.5 ±1.8	60.5 ±1.7	5.74 ±0.42M
GPT-6 Sol	37.3 [36.0, 38.6]	85.3 ±1.8	54.3 ±1.3	5.46 ±0.40M
Fable	45.9 [44.4, 47.4]	84.7 ±1.7	64.2 ±1.9	8.13 ±0.74M
Opus 5.5	**47.9** [46.4, 49.3]	85.5 ±1.5	**67.2** ±2.0	7.13 ±0.60M
GLM 5.3	40.0 [38.4, 41.5]	82.6 ±1.8	58.7 ±1.9	9.06 ±0.69M
Grok	37.9 [36.2, 39.6]	84.0 ±1.6	55.4 ±1.8	7.88 ±0.58M
DeepSeek V4 Pro	29.1 [27.8, 30.4]	84.0 ±1.7	42.3 ±1.2	6.33 ±0.46M
Kimi K3	43.0 [41.1, 44.9]	85.4 ±1.7	61.0 ±1.9	6.69 ±0.51M
Production library	45.1 [44.0, 46.2]	**85.6** ±2.6	62.1 ±2.8	9.83 ±1.90M
No library	32.9 [31.7, 34.0]	83.7 ±2.7	48.7 ±3.1	**3.28** ±0.66M

Astra

```
Command::new("inspect")
  .arg(Arg::new("verbose")
    .short('v')
    .action(ArgAction::Count))
  .arg(Arg::new("pids")
    .value_name("PID")
    .num_args(1..)
    .value_parser(
      ValueParser::u64()));
```

```
Command::new("inspect")
  .arg(Arg::new("verbose")
    .short('v')
    .action(ArgAction::Count))
  .arg(Arg::new("pid")
    .short('p').long("pid")
    .value_parser(
      ValueParser::from_str::<u32>())
    .required(true));
```

```
Command::new("inspect")
  .arg(Arg::new("verbose")
    .short('v')
    .action(ArgAction::Count))
  .arg(Arg::new("pid")
    .long("pid")
    .action(ArgAction::Append)
    .value_parser(
      ValueParser::u64()));
```

Fable

```
Command::new("procs")
  .arg(Arg::count("verbose")
    .short('v'))
  .arg(Arg::option("pid")
    .short('p')
    .uint()
    .multiple());
```

```
Command::new("proc")
  .arg(Arg::count("verbose")
    .short('v').global(true))
  .subcommand(Command::new(
      "inspect")
    .arg(Arg::positional("pid")
      .int().required(true));
```

```
Command::new("procinfo")
  .arg(Arg::positional("pid")
    .int()
    .required(true))
  .arg(Arg::count("verbose")
    .short('v'));
```

Figure 2: **Agents converge on the same designs.** README quick-starts from Astra (top) and Fable (bottom) across six `clirs` libraries each. **Teal**: all six; **orange**: Astra only; **violet**: Fable only.

Performance by implementer. Table 2 shows that all three implementers agree on the top three designers (Opus 5.5, Fable 5.1, GPT-6 Astra) and the last (DeepSeek V4 Pro), and none favors its own model family. Absolute scores differ across implementers, and averaging weights each equally.

3.2 Failure Taxonomy

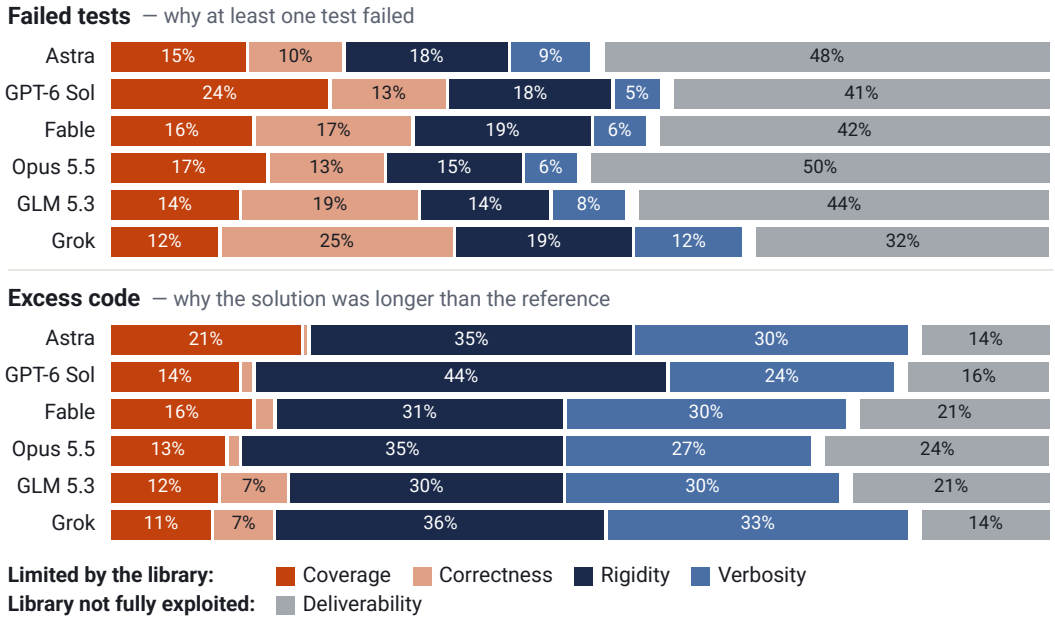


Figure 3: Primary failure category per designer. **Failed Tests** classifies why at least one test failed; **Excess Code** classifies why the solution was longer than the reference. Definitions are in Table 8.

Agent-written libraries resemble production libraries, yet implementers still write more code than the reference. We audit 810 partially passing solutions that exceed their references in code size, sampled from six designer configurations. We classify each failure by the smallest library change sufficient to prevent it. Appendix G gives the procedure and scope. The categories are *Coverage* (nothing close to the needed capability exists), *Correctness* (the capability exists but has a bug), *Rigidity* (it nearly fits but cannot be adapted to the task), *Verbosity* (it fits but requires excess code), and *Deliverability* (a simpler path exists, but the implementer did not find it).

In this sample, 82% of primary excess-code classifications fall under library limitations (Figure 3). Rigidity and Verbosity account for 64%, compared with 14% for Coverage. A common pattern we observe is that agent-written libraries implement only the exact core functionality the specification names. In `clirs`, 24 of the 33 written libraries do not easily implement long-option prefix inference, a capability expected under our full-featured-library brief; on problems that need it, lacking it is associated with scores 16.7 points lower than libraries that provide it. In an author analysis of agent-collected evidence, separate from the 810-cell audit, 41% of 13,562 hand-labeled lines that implementers rebuilt in 180 solutions redo functionality the library shipped but hid or broke.

3.3 How Do Agents Use Libraries?

LibraryUseBench measures how well agents use production libraries. Each of 8 models gets the production library and the minimal prompt (Listing 3), which only asks it to use the library and write little code. The GPT-5.6 Luna prompt and reasoning-effort comparisons in Table 3 instead use Codex. Figure 4 shows that natural library use scales with model capability.

Looking deeper, we vary prompt prescription (Appendix H; default in Listing 1) and reasoning effort for GPT-5.6 Luna (Table 3). Raising effort from low to high increases the pass rate from 51.4% to 84.5%, with Luna reading 15 library files instead of 2. The default prompt instead raises simplicity from 48.7 to 60.4 over the minimal one. Under the minimal prompt, 24% of trials never open the library and 36% of the reference's library symbols go unseen, versus 5% under the default. Luna searches by grepping API names it already expects, so capabilities it does not recall go unused (Appendix I). Even at the default setting, its simplicity remains well below the reference's.

Figure 4: **LibraryUseBench score vs. cost per problem.** All models run in mini-SWE-agent. Bars are 95% intervals.

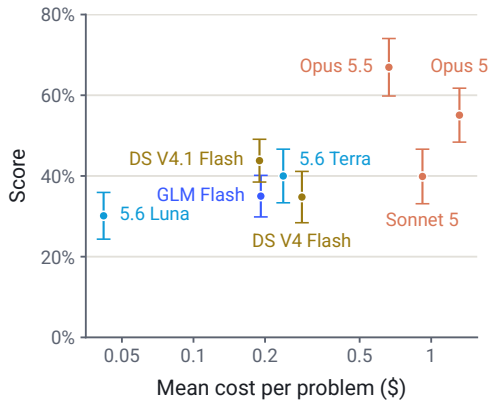


Table 3: **GPT-5.6 Luna (Codex, High) results with different reasoning efforts and prompts**. Effort rows use the default prompt; prompt rows use high effort.

Lever	Setting	Score	% Pass	Simplicity	$/prob.
Effort	Low	27.8	51.4	**83.2**	**$0.023**
	Med.	38.9	69.6	72.5	$0.055
Prompt	Min.	37.0	85.9	48.7	$0.082
	Low	36.5	**87.3**	46.4	$0.105
	Med.	44.3	85.7	57.3	$0.111
Default	High	**45.5**	84.5	60.4	$0.149

3.4 Do Agentic Design Patterns Work?

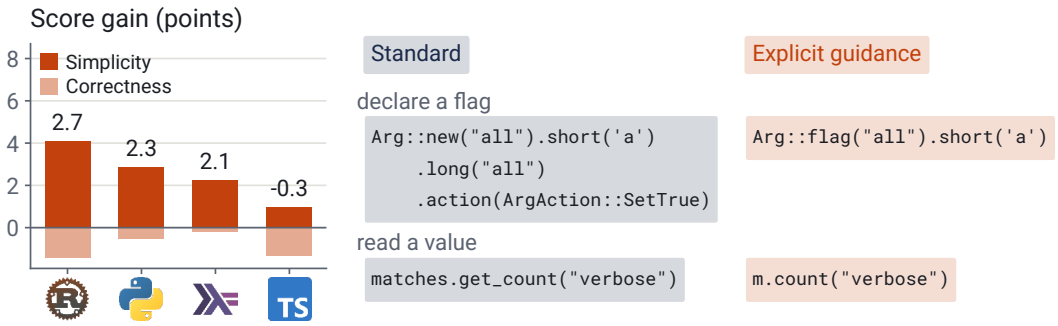


Figure 5: **Explicit guidance results**. Left: score gain per language, split into simplicity and correctness contributions (Appendix J). Right: `clirs` examples under each prompt.

We next test whether more prescriptive agent-first guidance changes the resulting libraries. GPT-6 Astra (Codex, high) designs each library with and without an explicit guidance prompt appended to the unchanged specification (Appendix J). The prompt states that only coding agents, scored on how little code they write, will use the library; it provides design guidance intended for agent users and combines consumer-first API sketches, runnable usage examples, and testing with subagents. We evaluate the combined intervention, not each component separately.

Guided libraries share fewer exported names with production (13.4% vs. 19.4%; Appendix J). In `clirs`, builder chains become single task-level calls (Figure 5, right). The score rises from 44.1 to 46.4, just below production (46.6) and standard-prompt Opus 5.5 (48.9). The score improves for all three implementers and three of four languages (Figure 5, left), mostly via shorter solutions. Guidance nearly doubles design cost ($8.24 vs. $4.51) with unchanged per-problem cost ($-\$0.01 \pm \$0.01$).

4 Related Work

Agents that write reusable code. Agents generate repositories (Ding et al., 2026; Liu et al., 2025; Chen et al., 2026; Zhang et al., 2026; Zhao et al., 2026; 2025; Hu et al., 2026; Peng et al., 2026; Yang et al., 2026), refactor code into libraries (Grand et al., 2024; Stengel-Eskin et al., 2024; Kovačič et al., 2025; Gautam et al., 2025; Jones et al., 2026; Thillen et al., 2026), work over long horizons (Orlanski et al., 2026; Asawa et al., 2026; Huang et al., 2026b; Ehrenberg et al., 2026; Cognition, 2026; Chu et al., 2026), and write tools for themselves or weaker models (Cai et al., 2024; Qian et al., 2023;

Yuan et al., 2024; Wang et al., 2024b). Most are graded by tests or task accuracy, which cannot see design: agents can pass nearly every test while leaving the requested library unused (Ma et al., 2026). The closest, ReGAL (Stengel-Eskin et al., 2024) and LATM (Cai et al., 2024), pass one model's code to another, but build small functions from solved tasks rather than design a library from an open specification. LibraryDesignBench grades design by downstream correctness and simplicity.

Evaluating libraries and their use. Software engineering judges an API by observing developers on fixed tasks, sometimes under competing designs (Ellis et al., 2007; Piccioni et al., 2013; Rauf et al., 2019), which finds that both API structure and documentation hinder developers (Robillard, 2009; Myers & Stylos, 2016), or by scoring the API surface (Scheller & Kühn, 2015), the code that reuse saves (Frakes & Terry, 1996), or complexity metrics (McCabe, 1976; Halstead, 1977; Campbell, 2018) that need not reflect what LLMs find hard (Xie et al., 2026; Patel et al., 2026). LLM benchmarks fix a library and score whether models call it correctly (Lai et al., 2023; Zhuo et al., 2025; Zan et al., 2022; Patil et al., 2024; Wang et al., 2024a; Jain et al., 2024; Chen et al., 2025). LibraryDesignBench keeps the user-study design with agents as users, and replaces completion time with correctness and code size against an expert reference; LibraryUseBench applies this measure to production libraries.

Designing for agents. Recent work studies how LLMs write and use code (Matias et al., 2026; He et al., 2026; Twist et al., 2026; Twist & Zhang, 2025; Watanabe et al., 2026) and argues code should be designed with agents as consumers (Wang et al., 2026; Borg et al., 2026; Patel et al., 2026), and agent-specific interfaces do help agents (Yang et al., 2024). Evaluations fix the consumer and vary its framework or documentation (Huang et al., 2026a; Wijaya et al., 2025), score agent-built tools one at a time (Kaliyev & Maryanskyy, 2026), or grade the agents that coding agents build (Shi et al., 2026). None varies the designer: LibraryDesignBench is, to our knowledge, the first to measure how well an agent designs a library for other agents.

5 Limitations

LibraryDesignBench evaluates the downstream value of a library under specified tasks, consumer configurations, and execution budgets. Scores characterize utility in that setting, not a consumer-independent library ordering. It measures tested correctness and the static size and complexity of consumer programs, not full library correctness, security, runtime efficiency, or maintainability. Reference programs normalize scores without prescribing the generated API, but are not uniquely optimal. The benchmark emphasizes workloads with opportunities for reuse, and its confidence intervals cover reruns of the fixed task set, not generalization to all library domains.

6 Conclusion

We introduced LibraryDesignBench, a benchmark that evaluates an agent-designed library only through how downstream agents use it, scoring the correctness and simplicity of their programs against expert references built on real production libraries. Across fifteen tasks in four languages, agents design libraries that help other agents, with Opus 5.5 scoring 4.9% above the production library, but they do so by reproducing the production library's design in eleven of fifteen tasks. Downstream agents then underuse these libraries: most excess code comes from rigid or hard-to-use interfaces rather than missing capabilities. Telling designers that only agents will use their library and having them test it with subagents yields libraries that copy fewer production designs and shrink downstream programs, yet the result still falls just short of the production library. Designing libraries for agents is therefore not the same as designing them for humans, and it remains an open problem. We hope LibraryDesignBench serves as a testbed for studying which interfaces, abstractions, and documentation help agents build on one another's code.

AI use statement

Generative AI is part of the benchmark and the object of study; the agents evaluated as library designers and implementers are listed in Appendix C. Beyond these experiments, we used generative AI tools to: generate benchmark data, where an agentic pipeline extracts, rewrites, and tests each downstream problem before expert review (Section 2.3); assist library experts in writing the reference

solutions; implement the evaluation harness, metric code, and analysis scripts; and collect and analyze data, including classifying failures in the audit (Appendix G), labeling shell commands in trajectories (Appendix I), and gathering evidence for the rebuilt-code analysis (Section 3.2). We have not used generative AI tools to formulate the research question or the scoring method. We used generative AI tools to help derive the standard errors and confidence intervals (Appendix A); translation is not applicable to this work. Additionally, we used generative AI tools to create figures and plotting code and to draft and edit the manuscript for clarity and concision. We have reviewed all AI-assisted work: library experts reviewed and rewrote every problem, AI-produced classifications were checked against manual labels, and the authors reviewed all AI-assisted derivations, code, analyses, figures, and text. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

Reproducibility statement

The anonymized repository linked in footnote 1 contains the evaluation harness and metric code and links to the benchmark data: the fifteen library specifications and the 242 downstream problems with their test suites and reference solutions. Section 2, Section 2.3, and Appendix F describe the task and how problems are built. Section 2.2 and Appendix A define the score, its aggregation, and the confidence intervals. Appendix B gives the static measures, pinned formatters (Table 6), and counting rules. Appendix C lists the models, harnesses, reasoning efforts, providers, and resource limits, and Appendices D, E, H, and J give the prompts. The failure audit and the library-search analysis are specified in Appendix G and Appendix I. Evaluated models are served by commercial APIs whose behavior may change over time, so exact rerun results may differ; we report costs at list prices as of September 2026.

References

Anthropic. System Card: Claude Fable 5.1 & Claude Mythos 5.1. https://www-cdn.anthropic.com/0339e6a7c5c7b87f5c07798616dc32c215d14235/Claude%20Fable%205.1%20&%20Claude%20Mythos%205.1%20System%20Card.pdf, September 2026a.

Anthropic. System Card: Claude Opus 5. https://www-cdn.anthropic.com/c5fbac3f0b1280a933ebd26d3cb8bb9f5bdeaf48/Claude%20Opus%205%20System%20Card.pdf, July 2026b.

Anthropic. System Card: Claude Opus 5.5. https://www-cdn.anthropic.com/fc1b44717c85dc068bc6ba5024219938094694bd/Claude%20Opus%205.5%20System%20Card.pdf, September 2026c.

Anthropic. System Card: Claude Sonnet 5. https://www.anthropic.com/claude-sonnet-5-system-card, June 2026d.

Parth Asawa, Christopher M. Glaze, Gabriel Orlanski, Ramya Ramakrishnan, Benji Xu, Asim Biswal, Vincent Sunn Chen, Frederic Sala, Matei Zaharia, and Joseph E. Gonzalez. Continual learning bench: Evaluating frontier ai systems in real-world stateful environments, 2026. URL https://arxiv.org/abs/2606.05661.

Markus Borg, Nadim Hagatulah, Adam Tornhill, and Emma Söderberg. Code for Machines, Not Just Humans: Quantifying AI-Friendliness with Code Health Metrics. In *Proceedings of the 2026 IEEE/ACM Third International Conference on AI Foundation Models and Software Engineering (FORGE)*, pp. 51–61. Association for Computing Machinery, January 2026. doi: 10.1145/3793655.3793722. URL https://arxiv.org/abs/2601.02200.

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large language models as tool makers. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2305.17126.

G. Ann Campbell. Cognitive complexity: an overview and evaluation. In *Proceedings of the 2018 International Conference on Technical Debt*, TechDebt '18, pp. 57–58, New York, NY, USA, 2018.

Association for Computing Machinery. ISBN 9781450357135. doi: 10.1145/3194164.3194186. URL https://doi.org/10.1145/3194164.3194186.

Jingyi Chen, Songqiang Chen, Jialun Cao, Jiasi Shen, and Shing-Chi Cheung. When LLMs Meet API Documentation: Can Retrieval Augmentation Aid Code Generation Just as It Helps Developers?, March 2025. URL https://arxiv.org/abs/2503.15231. arXiv: 2503.15231.

Silin Chen, Haoyi Teng, Xiaodong Gu, Yuling Shi, Jiale Huang, Yongpan Wang, Hongyu Zhang, and Haibing Guan. Repo0: Design-Driven Zero-to-All Code Generation, August 2026. URL https://arxiv.org/abs/2608.19854. arXiv: 2608.19854.

Evan Chu, Rajan Agarwal, Abishek Thangamuthu, Brendan Graham, Justus Mattern, Freeman Jiang, Paul Cento, Swarnim Jain, Mersad Abbasi, Mohammad Hossein Rezaei, George Wang, Alex Zhang, Simon Guo, Karina Nguyen, Danna Liu, Arash Bidgoli, Aditya Dalmia, Apoorv Dankar, Ashrut Vaddela, Calvin Chen, Keshav Kumar, Kushagra Vaish, Navid Pour, Rishyanth Kondra, Sagar Badiyani, Sidharth Giri, Snagnik Das, Soham Gaikwad, Syed Shah, Vagish Dilawari, and Vishal Agarwal. FrontierSWE. https://www.proximal.so/blog/frontierswe, 2026. Proximal Blog.

Cognition. Introducing FrontierCode: A coding eval that raises the bar for difficulty and quality. https://cognition.com/blog/frontier-code, 2026. Accessed: 2026-09-16.

DeepSeek-AI. DeepSeek-V4: Towards Highly Efficient Million-Token Context Intelligence. arXiv preprint arXiv:2606.19348, 2026. URL https://arxiv.org/abs/2606.19348.

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, Liya Zhu, Chenyang Zou, Ming Ding, Jianpeng Jiao, Jiaheng Liu, Minghao Liu, Qian Liu, Chongyang 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, 2026. URL https://arxiv.org/abs/2512.12730.

Henry Kiss Ehrenberg, Vincent Sunn Chen, Austin W. Hanjie, Karthik Narasimhan, Gabriel Orlanski, and Frederic Sala. Senior SWE-bench: Evaluating coding agents like senior engineers. https://snorkel.ai/blog/senior-swe-bench-evaluating-coding-agents-like-senior-engineers/, 2026. Accessed: 2026-09-16.

Brian Ellis, Jeffrey Stylos, and Brad Myers. The factory pattern in API design: A usability evaluation. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, pp. 302–312, 2007.

William Frakes and Carol Terry. Software reuse: Metrics and models. *ACM Computing Surveys*, 28 (2):415–435, 1996. doi: 10.1145/234528.234531.

Dhruv Gautam, Spandan Garg, Jinu Jang, Neel Sundaresan, and Roshanak Zilouchian Moghaddam. Refactorbench: Evaluating stateful reasoning in language agents through code. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=NiNIthntx7.

GLM-5 Team. GLM-5: From Vibe Coding to Agentic Engineering. arXiv preprint arXiv:2602.15763, 2026. URL https://arxiv.org/abs/2602.15763.

Gabriel Grand, Lionel Wong, Maddy Bowers, Theo X. Olausson, Muxin Liu, Joshua B. Tenenbaum, and Jacob Andreas. Lilo: Learning interpretable libraries by compressing and documenting code, 2024. URL https://arxiv.org/abs/2310.19791.

Maurice H. Halstead. *Elements of Software Science*. Operating and Programming Systems Series. Elsevier North-Holland, New York, NY, USA, 1977.

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.

Hao He, Courtney Miller, Shyam Agarwal, Christian Kästner, and Bogdan Vasilescu. Speed at the cost of quality: How cursor ai increases short-term velocity and long-term complexity in open-source projects. In *Proceedings of the 23rd International Conference on Mining Software Repositories*, MSR '26, pp. 181–193. ACM, April 2026. doi: 10.1145/3793302.3793349. URL http://dx.doi.org/10.1145/3793302.3793349.

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, April 2026. URL https://arxiv.org/abs/2604.06742. arXiv: 2604.06742.

Jintao Huang, Xiaomin Li, Gaurav Mittal, and Yu Hu. ADK Arena: Evaluating Agent Development Kits via LLM-as-a-Developer, June 2026a. URL https://arxiv.org/abs/2606.05548. arXiv: 2606.05548.

Wenqi Huang, Charley Lee, Leonard Tng, and Serena Ge. Deepswe: Measuring frontier coding agents on original, long-horizon engineering tasks, 2026b.

Nihal Jain, Robert Kwiatkowski, Baishakhi Ray, Murali Krishna Ramanathan, and Varun Kumar. On Mitigating Code LLM Hallucinations with API Documentation, July 2024. URL https://arxiv.org/abs/2407.09726. arXiv: 2407.09726.

R. Kenny Jones, Paul Guerrero, Niloy J. Mitra, and Daniel Ritchie. Shapelib: Designing a library of programmatic 3d shape abstractions with large language models, 2026. URL https://arxiv.org/abs/2502.08884.

Alibek Kaliyev and Artem Maryanskyy. Beyond Task Completion: A Verification-vs.-Conformance Gap in Tool-Evolving Agents, July 2026. URL http://arxiv.org/abs/2604.00392. arXiv:2604.00392 [cs.SE].

Kimi Team. Kimi K3: Open Frontier Intelligence. arXiv preprint arXiv:2607.24653, 2026. URL https://arxiv.org/abs/2607.24653.

Žiga Kovačič, Justin T Chiu, Celine Lee, Wenting Zhao, and Kevin Ellis. Refactoring Codebases through Library Design, May 2025. URL https://arxiv.org/abs/2506.11058. arXiv: 2506.11058.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. DS-1000: A natural and reliable benchmark for data science code generation. In *Proceedings of the 40th International Conference on Machine Learning*, 2023. URL https://arxiv.org/abs/2211.11501.

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, May 2025. URL http://arxiv.org/abs/2503.07010.

Yanuo Ma, Ben Kereopa-Yorke, and Ben Schultz. Building to the test: Coding agents deliver what you check, not what you requested, 2026. URL https://arxiv.org/abs/2606.28430.

Bruno Claudino Matias, Savio Freire, Juliana Freitas, Felipe Fronchetti, Kostadin Damevski, and Rodrigo Spinola. A survey on large language model impact on software evolvability and maintainability: the good, the bad, the ugly, and the remedy, 2026. URL https://arxiv.org/abs/2601.20879.

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

Evan Miller. Adding Error Bars to Evals: A Statistical Approach to Language Model Evaluations, November 2024. URL https://arxiv.org/abs/2411.00640. arXiv: 2411.00640.

Brad A. Myers and Jeffrey Stylos. Improving API usability. *Communications of the ACM*, 59(6): 62–69, 2016. doi: 10.1145/2896587.

OpenAI. GPT-5.6 System Card. https://deploymentsafety.openai.com/gpt-5-6, July 2026a.

OpenAI. GPT-6 Astra System Card. https://deploymentsafety.openai.com/gpt-6-astra, September 2026b.

OpenAI. GPT-6 Astra System Card, Appendix: GPT-6 Sol and GPT-6 Luna. https://deploymentsafety.openai.com/gpt-6-astra/sec:appendix-sol-luna, September 2026c.

Gabriel Orlanski, Devjeet Roy, Alexander Yun, Changho Shin, Alex Gu, Albert Ge, Dyah Adila, Nicholas Roberts, Frederic Sala, and Aws Albarghouthi. Slopcodebench: Benchmarking how coding agents degrade over long-horizon iterative tasks. *arXiv preprint arXiv:2603.24755*, 2026.

Shaswat Patel, Betty Li Hou, Arun Purohit, Kai Xu, Jane Pan, He He, and Valerie Chen. Is Agent Code Less Maintainable Than Human Code?, June 2026. URL https://arxiv.org/abs/2606.21804. arXiv: 2606.21804.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive APIs. In *Advances in Neural Information Processing Systems*, 2024. URL https://arxiv.org/abs/2305.15334.

Zhongyuan Peng, Dan Huang, Chuyu Zhang, Caijun Xu, Changyi Xiao, Shibo Hong, David Lo, Lin Qiu, Xuezhi Cao, Jiyuan He, and Yixin Cao. Icae-bench: Evaluating coding agents as interactive project builders, 2026. URL https://arxiv.org/abs/2607.21217.

Marco Piccioni, Carlo A. Furia, and Bertrand Meyer. An empirical study of api usability. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, pp. 5–14, 2013. doi: 10.1109/ESEM.2013.14.

Cheng Qian, Chi Han, Yi Fung, Yujia Qin, Zhiyuan Liu, and Heng Ji. CREATOR: Tool creation for disentangling abstract and concrete reasoning of large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, 2023. URL https://arxiv.org/abs/2305.14318.

Irum Rauf, Elena Troubitsyna, and Ivan Porres. A systematic mapping study of API usability evaluation methods. *Computer Science Review*, 33:49–68, 2019.

Martin P. Robillard. What makes APIs hard to learn? answers from developers. *IEEE Software*, 26 (6):27–34, 2009. doi: 10.1109/MS.2009.193.

Thomas Scheller and Eva Kühn. Automated measurement of API usability: The API concepts framework. *Information and Software Technology*, 61:145–162, 2015.

Quan Shi, Keshav Dhandhania, Karthik Narasimhan, and Victor Barres. τ^{τ}-Bench: An Environment for End-To-End, Realistic Agent Construction, September 2026. URL https://arxiv.org/abs/2609.04611. arXiv: 2609.04611.

Elias Stengel-Eskin, Archiki Prasad, and Mohit Bansal. Regal: Refactoring programs to discover generalizable abstractions, 2024. URL https://arxiv.org/abs/2401.16467.

Alex Thillen, Niels Mündler, Veselin Raychev, and Martin Vechev. Codetaste: Can llms generate human-level code refactorings?, 2026. URL https://arxiv.org/abs/2603.04177.

Priyansh Trivedi and Olivier Schmitt. Does code cleanliness affect coding agents? a controlled minimal-pair study. *ArXiv*, abs/2605.20049, 2026. URL https://api.semanticscholar.org/CorpusID:288653137.

Lukas Twist and Jie M. Zhang. A Study of Library Usage in Agent-Authored Pull Requests, December 2025. URL https://arxiv.org/abs/2512.11589. arXiv: 2512.11589.

Lukas Twist, Mark Harman, Don Syme, Joost Noppen, Helen Yannakoudakis, Detlef Nauck, and Jie M. Zhang. A study of LLMs' preferences for libraries and programming languages. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 331–351, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-395-1. doi: 10.18653/v1/2026.findings-acl.15. URL https://aclanthology.org/2026.findings-acl.15/.

Chong Wang, Kaifeng Huang, Jian Zhang, Yebo Feng, Lyuye Zhang, Yang Liu, and Xin Peng. LLMs Meet Library Evolution: Evaluating Deprecated API Usage in LLM-based Code Completion, June 2024a. URL https://arxiv.org/abs/2406.09834. arXiv: 2406.09834.

Shaolin Wang, Yi Mei, Haoyang Che, He Jiang, Shui Yu, and Ying Gu. From Human Interfaces to Agent Interfaces: Rethinking Software Design in the Age of AI-Native Systems, March 2026. URL https://arxiv.org/abs/2603.20300. arXiv: 2603.20300.

Zhiruo Wang, Daniel Fried, and Graham Neubig. TroVE: Inducing verifiable and efficient toolboxes for solving programmatic tasks. In *Proceedings of the 41st International Conference on Machine Learning*, 2024b. URL https://arxiv.org/abs/2401.12869.

Kan Watanabe, Tatsuya Shirai, Yutaro Kashiwa, and Hajimu Iida. What to Cut? Predicting Unnecessary Methods in Agentic Code Generation, February 2026. URL https://arxiv.org/abs/2602.17091. arXiv: 2602.17091.

Sandya Wijaya, Jacob Bolano, Alejandro Gomez Soteres, Shriyanshu Kode, Yue Huang, and Anant Sahai. ReadMe.LLM: A Framework to Help LLMs Understand Your Library, April 2025. URL https://arxiv.org/abs/2504.09798. arXiv: 2504.09798.

xAI. Model Card: Grok 4.6. https://media.x.ai/v1/website/card-4p6-4cd2dc57.pdf, August 2026.

Chen Xie, Xiaodong Gu, Yuling Shi, and Beijun Shen. Rethinking Code Complexity Through the Lens of Large Language Models, February 2026. URL https://arxiv.org/abs/2602.07882. arXiv: 2602.07882.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://arxiv.org/abs/2405.15793.

John Yang, Carlos E. Jimenez, Alexander Wettig, Shunyu Yao, Karthik Narasimhan, and Ofir Press. mini-swe-agent: A minimal and efficient software engineering agent suite. https://github.com/SWE-agent/mini-swe-agent, 2025. Accessed: 2026-09-16.

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. ProgramBench: Can Language Models Rebuild Programs From Scratch?, May 2026. URL https://arxiv.org/abs/2605.03546. arXiv: 2605.03546.

Lifan Yuan, Yangyi Chen, Xingyao Wang, Yi R. Fung, Hao Peng, and Heng Ji. CRAFT: Customizing LLMs by creating and retrieving from specialized toolsets. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2309.17428.

Daoguang Zan, Bei Chen, Zeqi Lin, Bei Guan, Yongji Wang, and Jian-Guang Lou. When language model meets private library. In *Findings of the Association for Computational Linguistics: EMNLP 2022*, pp. 277–288, 2022. URL https://arxiv.org/abs/2210.17236.

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?, May 2026. URL https://arxiv.org/abs/2605.07122. 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, June 2026. URL https://arxiv.org/abs/2606.10728. 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*, 2025. URL https://openreview.net/forum?id=MMwaQEVsAg.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. BigCodeBench: Benchmarking code generation with diverse function calls and complex instructions. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://arxiv.org/abs/2406.15877.

APPENDIX

A BENCHMARK AGGREGATION AND REPORTING

Benchmark aggregation. To evaluate the designer rather than a single artifact, we independently repeat the design phase $K = 3$ times for each of the $\mathcal{T}$ benchmark tasks. We evaluate each resulting library on its task's problems using the same fixed implementer set, with fresh downstream executions. We average all generations rather than selecting the best, reducing the influence of an unusually successful or unsuccessful run. The production-library and no-library settings have no design phase; for them, we repeat the evaluation phase $K = 3$ times, and k indexes these repetitions.

Let $L_{p,k}$ be the library generated for task p on run k, and let

$$S_{p,k} = \text{score}(L_{p,k}), \qquad \bar{S}_p = \frac{1}{K}\sum_{k=1}^{K} S_{p,k}. \tag{3}$$

Here, $S_{p,k}$ retains the average over problems and implementers defined in Equation 1. We then give each library-design task equal weight:

$$\widehat{S} = \frac{1}{\mathcal{T}}\sum_{p=1}^{\mathcal{T}} \bar{S}_p = \frac{1}{\mathcal{T}K}\sum_{p=1}^{\mathcal{T}}\sum_{k=1}^{K} S_{p,k}. \tag{4}$$

This prevents tasks with more downstream problems from receiving greater benchmark weight.

Rerun standard errors. To quantify how much the reported score would fluctuate, given the stochastic nature of agentic evaluation, we formulate a standard error for agent-to-agent evaluations. Each independently generated library together with its complete downstream evaluation constitutes one observation.

These observations need not be identically distributed across tasks as different tasks have disparate expected scores and execution variances. We therefore estimate variability *within* each task, treating tasks as fixed strata rather than measuring deviations around a single overall mean. For each task, the sample variance across library runs is

$$s_p^2 = \frac{1}{K-1}\sum_{k=1}^{K}(S_{p,k} - \bar{S}_p)^2. \tag{5}$$

Assuming independent, identically distributed repetitions within each task and independence across tasks, the estimated standard error of the benchmark mean is

$$\widehat{\text{SE}}_{\text{run}}\left(\widehat{S}\right) = \frac{1}{\mathcal{T}}\sqrt{\sum_{p=1}^{\mathcal{T}} \frac{s_p^2}{K}}. \tag{6}$$

The squared expression is an unbiased estimator of the variance of Equation 4, provided the run scores have finite variance. Each library's deviation is measured relative to its own task mean, so stable differences between tasks do not contribute.

We retain dependence within a library evaluation by computing $S_{p,k}$ before estimating its variance. For example, an architectural defect can hurt several problems or implementers simultaneously. These shared effects contribute to the variance of the complete library score, so we do not treat the downstream programs as independent library observations. This follows the principle of retaining related evaluations together when estimating uncertainty (Miller, 2024).

Confidence intervals. We report approximate 95% confidence intervals for the designer's expected score under repeated execution of this fixed evaluation:

$$\mu = \frac{1}{\mathcal{T}} \sum_{p=1}^{\mathcal{T}} \mathbb{E}[S_{p,1}].$$

Because the task-specific variances are estimated from a small number of runs, we use a Student-t interval with Welch–Satterthwaite effective degrees of freedom:

$$\widehat{\nu} = \frac{\left(\sum_{p=1}^{\mathcal{T}} s_p^2/K\right)^2}{\sum_{p=1}^{\mathcal{T}} \frac{(s_p^2/K)^2}{K-1}}. \tag{7}$$

The reported interval is

$$\widehat{S} \;\pm\; t_{\widehat{\nu},\,0.975}\, \widehat{\mathrm{SE}}_{\mathrm{run}}\Big(\widehat{S}\Big)\,, \tag{8}$$

where $t_{\nu,\,0.975}$ is the 97.5th percentile of a Student-t distribution with ν degrees of freedom. The interval is a model-based approximation motivated by approximately normal within-task run-score distributions. With only three generations per task, nominal coverage is not guaranteed; the interval reflects stochastic variation in both phases on this fixed benchmark.

Descriptive standard errors. The $\pm$ values reported for pass rate, simplicity, cost, and tokens are standard errors of the mean clustered by task, $\widehat{\mathrm{SE}}_{\mathrm{cl}}(\cdot)$. They describe variation across problems and runs rather than the rerun uncertainty of the score. A reported difference between two conditions' descriptive means (e.g., the change in cost per problem) combines the two conditions' clustered standard errors in quadrature. Score differences between conditions instead combine the two conditions' rerun standard errors, $\widehat{\mathrm{SE}}_{\mathrm{run}}(\cdot)$, in quadrature.

B Static Measurement Details

Pipeline. We first format each submission with a pinned formatter set to 80 columns, so all code has the same layout and line counts are fair. Code that fails to format scores zero. We then parse each source file with tree-sitter (0.25.2) and count every metric in one pass over the syntax tree. Build output, dependencies, and symlinks are skipped. If the parser cannot read part of a file, we still measure the rest; the formatter decides whether code is valid.

Metrics.

- **SLOC**: nonblank lines, not counting comments or Python docstrings.
- **Cyclomatic complexity**: $+1$ for each function, each branch, and each `&&` or `||`.
- **Cognitive complexity**: each branch adds 1 plus its nesting depth, and code inside it is one level deeper. Each `&&` or `||` adds 1. Functions do not add depth.
- **Halstead volume**: $N \log_2 \eta$, where N is the number of operators and operands and η is the number of distinct ones across the whole workspace. Operands are names and literals.

Python (Ruff 0.16.6). `if`, `for`, `while`, `except`, ternaries, and the `for`/`if` parts of comprehensions are branches. `elif` adds 1 to each complexity, with no nesting cost. `match` adds only to cognitive complexity, and each `case` except `_` adds 1 to cyclomatic. `assert` adds 1 to cyclomatic. A `lambda` does not count as a function.

JavaScript and TypeScript (Prettier 3.9.6). Both use the same rules. `if`, loops, `catch`, each `switch` case, and ternaries are branches. An `else if` counts as an `if` nested inside the one before it, so long chains cost more. Every function counts, including arrow functions and methods. `??` adds 1 to cyclomatic only. A TypeScript submission also counts JavaScript files it uses.

Rust (rustfmt 1.9.0). `if`, `for`, `while`, `loop`, and `match` are branches. A `match` counts once; its arms add nothing. A labeled `break` or `continue` adds 1 to cognitive. A function that calls itself adds 1 to cognitive, once. Code inside macros and the `?` operator counts as zero.

Haskell (Fourmolu 0.20.1.0). Each equation of a function adds 1 to cyclomatic. `if` is a branch. A `case` adds to cognitive like a branch and adds 1 to cyclomatic for each alternative after the first. Guards work like an if/else-if chain: each guard adds 1 to cyclomatic; the first guard adds 1 plus nesting depth to cognitive, and each later guard, including `otherwise`, adds 1. `<|>` counts like `||`.

C Setup Details

Agents. All agents run in mini-SWE-agent (Yang et al., 2025) unless otherwise specified, at "High" reasoning effort. Designers: GPT-5.6 Sol (Codex) (OpenAI, 2026a), GPT-6 Sol (OpenAI, 2026c), GPT-6 Astra (OpenAI, 2026b), Opus 5.5 (Anthropic, 2026c), Fable 5.1 (Anthropic, 2026a), GLM 5.3 (GLM-5 Team, 2026), Grok 4.6 (xAI, 2026), DeepSeek V4 Pro (DeepSeek-AI, 2026), and Kimi K3 (Kimi Team, 2026); Fable 5.1 also runs in Claude Code (CC) and GPT-6 Astra in Codex. Implementers: GPT-5.6 Luna (Codex) (OpenAI, 2026a), GLM 5.3 Flash (GLM-5 Team, 2026), and DeepSeek V4.1 Flash (DeepSeek-AI, 2026), one attempt per problem with the same prompt. LibraryUseBench (Section 3.3) evaluates these three implementer models plus Opus 5.5 (Anthropic, 2026c), Opus 5 (Anthropic, 2026b), Sonnet 5 (Anthropic, 2026d), GPT-5.6 Terra (OpenAI, 2026a), and DeepSeek V4 Flash (DeepSeek-AI, 2026) as implementers on the production library, all evaluated in mini-SWE-agent. The GPT-5.6 Luna prompt and reasoning-effort comparisons (Table 3) instead use Codex. OpenAI, Anthropic, and xAI models use first-party APIs; GLM uses OpenRouter; DeepSeek and Kimi use Prime Inference. Costs use official list prices as of September 2026.

Environment. Harbor (Harbor Framework Team, 2026) runs one Docker image per task, shared by both phases and every library condition (Table 4). Preinstalled packages are not target-domain libraries, and health checks block substituting a human-written library. The library is mounted at `/library` and the agent works in `/workspace`. The agent network is limited to model-provider APIs. Library specifications get packaging instructions appended (Appendix D). Limits are in Table 5; pinned formatters are in Table 6.

Table 4: Per-task Docker images, shared by both phases and all library conditions. Counts are tasks.

Scope	Toolchain	Dependencies
All (15)	`bash`, `git`, `curl`, `jq`, `rg`, Node 22.14, Python 3.12, `uv` 0.9.5	–
Python (6)	`ruff`	Pinned `requirements.txt` in `/workspace/.venv`; `uv` cache warmed
TypeScript (1)	`tsc`, `tsx`, `prettier`, Chromium	Pinned `package.json`
Rust (5)	`cargo`, `rustfmt`, `clippy`; Rust 1.85–1.91	Prefetched `Cargo.lock`; `CARGO_NET_OFFLINE`
Haskell (3)	GHC 9.8.4, `cabal`, `fourmolu`	Prefetched frozen Cabal closure

Table 5: Harness-enforced limits per phase. The agent budget is wall-clock inside the container. Design runs against wall-clock alone; a problem ends at whichever of its two budgets binds first.

	Design Phase (design)	Evaluation Phase (one problem)
Agent wall-clock	4 hours	60 minutes
Verifier wall-clock	15 minutes	5–60 minutes, set per problem
Sandbox	4 CPU, 8 GB	2 CPU, 4 GB
Spend cap	None	$2.50
Attempts	1 ($K = 3$ independent runs per task)	1
Agent network	Model-provider API allowlist only	Model-provider API allowlist only
Verifier network	Disabled	Disabled

D Library Packaging Instructions

Every Design Phase specification ends with a general-instructions block appended to the capability brief in Figure 1. The block asks for all functionality users would reasonably expect of such a library,

Table 6: Pinned normalizers and grammars used for static measurement. The same versions are applied to the optimized references and to every generated program. Measurement uses its own pinned Rust 1.98.1 toolchain for `rustfmt`, separate from the per-task build toolchains in Table 4; TypeScript problems additionally load the JavaScript grammar for embedded sources.

Language	Formatter (pinned)	Config	`tree-sitter` grammar
Python	`ruff format` 0.16.6	`ruff.toml`	`python`
TypeScript	`prettier` 3.9.6	`prettier.json`	`typescript`
Rust	`rustfmt` 1.9.0-stable	`rustfmt.toml`	`rust`
Haskell	`fourmolu` 0.20.1.0	`fourmolu.yaml`	`haskell`

states that it will be used primarily by coding agents and the senior engineers who review their work, and fixes a language-specific package layout so the Evaluation Phase harness can install the library without guessing. Each layout names the package manager and package name and requires a lockfile limited to the offline dependencies the block lists. Where a single build command exists, the block states it and the library must pass it offline.

- **Python.** `pyproject.toml` with `[project].name`, managed by `uv`, with a `uv.lock`.
- **TypeScript.** `package.json` with `name` and a `package-lock.json`; the package ships the runtime files its public entry points need and TypeScript declarations for its public interface.
- **Rust.** `Cargo.toml` with `[package].name` and a `Cargo.lock`; builds with `cargo build --manifest-path /workspace/Cargo.toml --offline`.
- **Haskell.** A top-level `.cabal` file naming the library and exposing its modules, plus `cabal.project` and `cabal.project.freeze`; builds with `cabal build all --offline` and as a source dependency of another Cabal project.

The block does not constrain the public interface, the module decomposition, or the abstractions. That design freedom is what LibraryDesignBench measures.

E MAIN PROMPTS USED

Listing 1: Implementer prompt.

```
<task instruction>

## Library Rules

Your solution is a thin adapter around `<library>`. It is judged on how
little code sits on top of the library, so every operation the library
can
carry, the library carries.

- `<library>` is installed. Its source, examples, tutorials, and docs are
 in `/library` (read-only).
- Reach for the primitive that does the whole operation (the parser,
validator, pipeline, runner), not its pieces. Importing constants, error
types, or small helpers while hand-rolling the operation is not using the
 library.
- If the library's default behavior differs from the task, configure or
extend the library. Reimplementing is the last resort, and only after a
search confirms the library lacks it.
- Handle exactly the validation the task describes.
- Other available dependencies: <dependencies>. Use them for work outside
 the library's domain.
- You have one hour.

## Workflow

1. **Map the task onto the library.** Read `/library` in this order:
README and docs, then examples, then grep the source for each concept the
 task names. Done when every requirement in the task is paired with the
library entry point that carries it, or with "not provided" after a
search.
2. **Build the program in `/workspace`** by calling those entry points.
3. **Audit.** For each function, loop, branch, and check you wrote, name
the library call that replaces it and use that instead, or note why the
library lacks it. Done when every remaining hand-written line has a
reason.
4. **Run the task's sample inputs.** Done when each produces the
described output. Sample runs are enough; skip test suites.
5. Submit.
```

Listing 2: No-library implementer prompt.

```
<task instruction>

## Rules

Your solution is judged on how little code it takes, so keep it as small
and
direct as the task allows.

- Available dependencies: <dependencies>. Use them for work outside the
task's core domain (CLI parsing, serialization, etc.). Nothing else is
installed.
- Handle exactly the validation the task describes.
- You have one hour.

## Workflow

1. **Build the program in `/workspace`.**
2. **Run the task's sample inputs.** Done when each produces the
described output. Sample runs are enough; skip test suites.
3. Submit.
```

F Benchmark Problems

Table 7 lists the fifteen library-design problems, the production library each one's reference solutions are written against, and the number of Evaluation Phase problems built for it. The library name is the name the designer agent is told to package under, and is the name used throughout this paper.

Table 7: The fifteen LibraryDesignBench library-design problems.

Library	Language	Production library	Domain	Problems
`canon`	Python	`pydantic`	Schema validation	14
`pyda`	Python	`pandas`	Dataframes	21
`roadkill`	Python	`uxsim`	Traffic simulation	16
`sapi`	Python	`fastapi`	HTTP API framework	11
`simu`	Python	`simpy`	Discrete-event simulation	13
`uglypie`	Python	`beautifulsoup4`	HTML parsing	11
`heretical`	TypeScript	`dompurify`	HTML sanitization	22
`arazu`	Rust	`rapier`	Rigid-body physics	22
`clirs`	Rust	`clap`	CLI construction	13
`npr`	Rust	`ndarray`	N-dimensional arrays	20
`rubberband`	Rust	`tantivy`	Full-text search	20
`weft`	Rust	`chumsky`	Parser combinators	14
`plaid`	Haskell	`megaparsec`	Parser combinators	15
`umami`	Haskell	`unagi-chan`	Concurrent channels	20
`wapi`	Haskell	`servant`	Typed web APIs	10
Total				**242**

G Taxonomy

Audit protocol. For each generated library from the 6 mini-SWE-agent designers, we sample 3 downstream cells (one implementer's solution to one problem) that both failed at least one test and wrote more code than the reference; every sampled cell also passed some tests, and cells that passed every test are not audited. One GPT-5.6 Luna agent with high reasoning audits each cell from the library, the implementer's trajectory and solution, the test results, and the reference solution. It describes the best as-shipped path without executing it and returns up to three causes per symptom, largest first. We report the first as the cell's primary classification, giving 810 excess-code and 810 failed-test classifications from the same cells.

Scope. Each cell is audited once by one model, from the same family as one implementer, without human labels or an agreement check. The auditor reasons about the best as-shipped path rather than



Figure 6: **Agent-written libraries outperform production libraries on a subset of tasks.** Difference in score per task compared with that of the production library. The rightmost column is the no-library setup; outlined cells score below no library.

running it. It attributes every failure to the library change that would have prevented it, including an implementer's own task-logic errors. The shares describe partially passing cells, not every downstream cell.

Table 8: Failure-taxonomy categories and leaves; Appendix G gives the classification procedure.

Category	Leaf	Definition
Limited by the library: no path through the library as shipped does better.		
Coverage	Absent operation	No operation or documented composition performs this reusable domain computation, and none does nearly this. The fix is a new operation.
Correctness	Contract violation	The library returns wrong output on a legitimate input.
	Misleading diagnostic	An error pointed away from the actual cause, and the implementer followed it.
	Performance defect	The library path is too slow for the problem's limits.
Rigidity	Fixed policy	An operation hard-codes how it works, such as ordering, rounding, error handling, or output format, with no parameter that selects what the problem needs.
	Closed representation	The library's data model cannot represent the problem's value, field, or variant, so the implementer builds parallel types.
	Monolithic operation	An operation bundles several steps; the implementer needs one of them or a small variant, and the inner steps are not exposed.
	Excluded scope	A primitive excludes, by design, what the problem feeds it, such as a mode, encoding, or input layout.
Verbosity	Verbose interface	Using the library takes at least as much code as writing the logic by hand.
	Unpackaged composition	Reusable multi-step wiring around an operation that no helper packages.
	Shape mismatch	Mechanical conversion between the library's inputs or outputs and the shapes the problem needs.
	Repeated declaration	The same setting must be declared at several sites, with no shared place for it.
Library not fully exploited: a path through the library as shipped removes the code or fixes the failure.		
Deliverability	Not surfaced	The implementer's reads never returned the capability, because of where it lives, what it is called, or what is exported.
	Not recognizable	The implementer saw it, but its description does not express the need in the problem's terms.
	Not salient	The implementer saw an adequate description, but it did not reach the decision: deep in a long output, cut off, or missed by a later search.
	Incomplete contract	The implementer found the capability but not a fact needed to use it correctly, such as a precondition, default, or required setting.
	Undistinguished alternatives	Several adjacent options were available, and nothing said which one fits.

H Library Usage Prompts

Listing 3: Minimal-prescription implementer prompt.

```
<task instruction>

## General Instructions

- You **must** use `<library>` (Source is at `/library`, installed for you
 already).
- Write as little code as possible.
- These are all of the dependencies available to you: <dependencies>.
- You have one hour.
```

Table 9: Failure-taxonomy leaf counts per designer, over the standardized subset of **Failed tests** and **Excess code** cells classified in Section 3.2. Each designer contributes 135 cells per symptom stratum. Leaf definitions are in Table 8.

		Failed tests							Excess code						
Category	Leaf	Astra	GPT-6 Sol	Fable	Opus 5.5	GLM 5.3	Grok	All	Astra	GPT-6 Sol	Fable	Opus 5.5	GLM 5.3	Grok	All
Limited by the library															
Coverage	Absent operation	20	32	21	23	19	16	131	28	19	21	17	16	15	116
Correctness	Contract violation	9	17	22	17	25	33	123	1	2	3	2	9	9	26
	Misleading diagnostic	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Performance defect	5	0	1	0	1	1	8	0	0	0	0	1	0	1
Rigidity	Fixed policy	17	18	19	17	14	19	104	38	41	31	32	32	41	215
	Closed representation	3	2	3	0	0	1	9	0	1	2	1	0	1	5
	Monolithic operation	0	1	0	1	2	2	6	3	6	4	5	1	4	23
	Excluded scope	4	3	4	2	3	4	20	6	12	5	9	7	2	41
Verbosity	Verbose interface	3	2	2	3	4	5	19	9	10	12	9	10	11	61
	Unpackaged composition	6	2	6	2	7	8	31	27	22	26	26	29	32	162
	Shape mismatch	3	3	0	3	0	3	12	4	1	3	0	1	1	10
	Repeated declaration	0	0	0	0	0	0	0	0	0	0	1	0	0	1
Library not fully exploited															
Deliverability	Not surfaced	0	0	0	1	0	1	2	0	0	0	0	1	0	1
	Not recognizable	11	12	13	9	14	8	67	9	12	7	12	14	1	55
	Not salient	17	4	12	9	12	7	61	6	3	13	7	6	6	41
	Incomplete contract	35	35	28	37	29	24	188	0	4	3	3	3	6	19
	Undistinguished alternatives	2	4	4	11	5	3	29	4	2	5	11	5	6	33
Limited by the library		70	80	78	68	75	92	463	116	114	107	102	106	116	661
Library not fully exploited		65	55	57	67	60	43	347	19	21	28	33	29	19	149
Cases		135	135	135	135	135	135	810	135	135	135	135	135	135	810

Listing 4: Low-prescription implementer prompt.

```
<task instruction>

## General Instructions

- You **must** use `<library>` as much as possible in your solution to
minimize its size.
    - `<library>` has already been installed for you. Its raw source,
examples, tutorials, and docs are in `/library`.
- These are all of the dependencies available to you: <dependencies>.
- Your programs only need to handle what is described above.
- You have one hour.

**Suggested Workflow:**
1. Read the examples/tutorials/docs/etc in `/library` before writing code.

2. Build the program in `/workspace`.
3. Run it with examples to ensure it works. You do not need to write test
 suites.
4. Submit.
```

Listing 5: Medium-prescription implementer prompt.

```
<task instruction>

## General Instructions

- Your solution is a thin adapter around `<library>`. Every line you
write by hand that `<library>` could have carried makes the solution
worse.
    - `<library>` has already been installed for you. Its raw source,
examples, tutorials, and docs are in `/library`. Treat it as read-only.
    - Use the highest-level interface of `<library>` that fits the task.
Importing constants, error types, or small utilities does not count as
using the library when it exposes a broader primitive for the same work.
    - Search `/library` before choosing an interface. Only hand-write
operations you have confirmed are not provided by `<library>`.
- These are all of the dependencies available to you: <dependencies>. Use
 them for work outside the library's domain.
- Your programs only need to handle what is described above. Only handle
the validation described!
- You have one hour.

**Suggested Workflow:**
1. Read the examples/tutorials/docs/etc in `/library` before writing code.

2. Build the program in `/workspace`.
3. Reread every function, loop, branch, and check you wrote and ask
whether `<library>` already does it. If it does, delete your version and
call the library. Ideally this finds nothing because you built on the
library from the start.
4. Run it with sample inputs to ensure it works. You do not need to write
 test suites.
5. Submit.
```

I How Agents Search Libraries

We classify every library-touching shell command in each LibraryUseBench and GPT-5.6 Luna trajectory as a search (a grep-style search over the library), documentation read, example read, source read, listing, or introspection call, and parse each solution's library imports. This covers 14 runs, 10,164 trials, and 412,802 shell commands. Command categories agree with manual labels on 59 of 60 spot-checked commands, and import extraction on 30 of 30. We count a reference-solution symbol as *seen* when it appears in the output of a library read, which makes seen rates an upper bound.

Agents verify the API they remember. Under the minimal prompt, 24% of Luna trials never touch the library and another 24% only list it or print its version. Every skip is on a well-known library (e.g., 85% of `bs4` and 76% of `pandas` trials), and 92% of those solutions still import it from memory. When agents search, 64–87% of grep patterns are identifier-shaped (e.g., `RevoluteJointBuilder`, `fn map_with`), and only 16–20% share a word with the task. The grep→source→grep loop appears in 38–80% of trials in every run. Across LibraryUseBench models the loop is the same and only its volume differs: Opus 5.5 reads 1.4 library files per trial and DeepSeek V4.1 Flash reads 4.9.

Prescription moves reading up front. Table 10 shows that more prescriptive prompts add documentation reading before the first write and read more of the library, while the share of grep and source commands stays flat. Symbols the agent never saw become symbols it uses; seen-but-unused symbols stay at 17–22% from the low prompt upward. The audit step of the default prompt is mostly stated rather than performed: 56% of trials mention an audit in their reasoning after the first clean run, but only 40% read the library again, and fewer than 11% read it within three commands of a failing run.

Table 10: GPT-5.6 Luna library interaction by prescription level (high effort).

	Minimal	Low	Medium	High (default)
Never touch library (%)	23.7	0.0	0.0	0.1
Docs/examples read before first write (%)	13	88	87	100
Library grep before first write (%)	44	88	98	100
Distinct library files read	3.5	7.5	10.0	14.8
Distinct grep terms	12	20	33	46
Grep share of library commands	.32	.27	.33	.29
Source share of library commands	.30	.25	.32	.30
Reference symbols used (%)	55	64	74	78
Reference symbols seen, not used (%)	9	22	18	17
Reference symbols never seen (%)	36	14	8	5
Audit language after first clean run (%)	4	3	21	56
Library read after first clean run (%)	25	25	35	40

Reasoning effort sets search depth. Table 11 shows that low effort stops at listings and documentation. It also concludes early that the library lacks a capability: 88–91% of first "library lacks X" claims come before the first write, and in a 150-trial hand-labeled sample only 22–38% follow a grep for that capability. In one `tantivy` problem, a low-effort grep for `Stemmer` piped through `head -100` cut off the tokenizer registrations, and the agent concluded that stemming is not built in. It scored 0, while all six medium- and high-effort attempts used the default `en_stem` tokenizer and scored 79–96.

Table 11: GPT-5.6 Luna library interaction by reasoning effort (default prompt).

	Low	Medium	High
Distinct library files read	2.0	5.7	14.8
Library commands per trial	3.1	5.3	12.2
Any library grep (%)	45	60	91
Any library source read (%)	23	50	81
Only listings or docs (%)	42	18	0.4
Return to library after first write (%)	22	30	47
Agent steps	15	27	47

Agents copy idioms rather than search for short names. Only 4% of trials grep for a prelude, `__all__`, or re-exports, and 62% of greps for export declarations name a single symbol. Solutions still import from the library root or prelude in 91% of Python, 70% of Rust, and 47% of Haskell cases, because they copy the documented idiom: `import pandas as pd` in every `pandas` trial, `prelude::*` in 83% of `rapier` and 93% of `chumsky` trials. Deeper reading instead produces deeper imports. Agents import a symbol from where they found it (e.g., `bs4.element.Tag`), and deep imports where a short path exists rise from 4% to 21% of Rust trials between low and high effort.

J Explicit Guidance Prompt

The prompt calls this agent-oriented design style *neuralese*.

Reporting definitions. *Exported-name overlap* is the share of the production library's exported names that a generated library reproduces verbatim, averaged over a task's libraries and then over tasks. The *simplicity and correctness contributions* in Figure 5 split each paired cell's score change (the same implementer and problem under both prompts) into the part due to the change in simplicity and the part due to the change in test-pass fraction, using a symmetric two-factor split of the per-problem product in Equation 1. Contributions are averaged over tasks and then over languages with equal weight. Under this split, the guidance prompt gains 2.5 points from simplicity and loses 0.9 from correctness.

Listing 6: Explicit guidance prompt: the agent-oriented library-design condition of Section 3.4.

```
{{instruction}}

## Who this library is for

Humans do not need to understand this library at all, only agents. Fresh
coding agents. Each one gets a task in this domain, has `{{library_name}}
` installed with its docs, and is scored on how few tokens of code it
writes on top of the library while passing hidden tests. The library is a
 channel between you and that agent: you compress the domain into an API,
 the agent decompresses its task into a handful of calls. Every token the
 agent still has to write is a token your channel failed to carry.

Design in **neuralese**: the form two models would settle on if they only
 had to talk to each other. Optimise token economy for a model, not
legibility for a person. Judge every design choice by one question: *what
 would an agent prefer?*

## What an agent prefers

- One verb per intent that carries the whole task: parse this document,
resolve these references, render that report. A model states its intent
in one line and wants one call that matches it.
- Names that are the intent, arguments that are the task's own nouns,
results that are the shape the task asked for.
- Defaults that already match the common case, so the common program has
no configuration at all; the uncommon case is one keyword away.
- Edge cases, validation, ordering, formatting and diagnostics inside the
 call. The agent writes the happy path and gets the correct program.
- Dense examples over prose. A model finds the example nearest its task
and copies it; it reads reference docs only when no example fits.
- Big flat surfaces over layers. Human decomposition, small composable
pieces, builders, class hierarchies, configuration objects and
abstractions earn a place only where the agent's program gets shorter
with them than without.

## Workflow

1. **Write the consumer's programs first.** From the example usages above
 and the tasks a library of this kind exists for, write ten to fifteen
distinct downstream tasks as the program a model would most want to write,
 in `/workspace/SKETCHES.md`: three to eight lines each, calling
functions that do not exist yet. Done when the set spans input parsing,
the core operations, output shapes and error paths, and no sketch holds a
 loop, branch or helper the library could own.
2. **Design the API from the sketches.** Every function a sketch calls is
 public API with that name and signature. Done when each sketch type-
checks against the design.
3. **Implement**, using only these dependencies: {{libraries}}.
4. **Ask the agents.** If you can spawn subagents, do it: for each sketch,
 hand a subagent only the downstream task text plus the library as
installed and its README, with no memory of your design, and have it
write the program. Where its program is longer than the sketch, guesses a
 name wrong, or has to read reference docs, fix the library, then ask
again. If you cannot spawn subagents, do the same from a clean context
yourself, task text and README only. Done when a fresh agent lands on the
 sketch without help, for every sketch.
5. **Freeze the examples.** Each sketch, unchanged, becomes a runnable
file under `/workspace/examples/` and runs on realistic input. Done when
every example runs.
6. **Write `/workspace/README.md`** for the consumer: the examples first,
 each with one line naming the task it solves, then the reference, then
packaging.
```

```
7. **Package** as the task instructions specify, build offline, and
submit.
```

K TIMEOUTS

Of the 28,314 Evaluation Phase trials (one implementer solving one problem with one library), 2.3% ended through budget exhaustion or library-installation failure: the agent ran out of time or budget (1.7%) or the authored library failed to install (0.6%). We keep every such trial; when the agent runs out of time or budget, we grade whatever it left behind. Dropping these trials instead raises the mean score by at most 1.2 points per arm and does not change the ordering of the no-library, agent-authored, and production-library arms.