SHAPINGBENCH: CAN CODING AGENTS REPRODUCE MATURE SOFTWARE BEHAVIOR BEYOND EXPLICIT REQUIREMENTS?

Anonymous authors
Paper under double-blind review

ABSTRACT

Coding agents are becoming increasingly capable of implementing software from high-level requirements, often expressed in natural language. However, existing benchmarks mainly evaluate behavior that is explicitly specified in the underlying task. We instead study how much of the behavior accumulated in mature software an agent can realize when those expectations are never individually stated. We introduce **ShapingBench**, a benchmark spanning 9 software domains, 45 mature open-source implementations, and 14,537 executable behavioral contracts. Behaviors are recovered from five independently developed implementations per domain and organized by how widely they recur across them. Behaviors shared by all five form the *Shared Core*, while the remaining behaviors form the *Variable Surface*. More than half of the Variable Surface appears in at least two reference implementations. We evaluate 4 LLMs across 3 agent frameworks, covering 10 model and framework combinations and 90 development trajectories. Behavioral coverage decreases consistently with cross-implementation recurrence. More importantly, no evaluated configuration fully realizes the *Shared Core*, even though every behavior in this subset is reproduced by all five reference implementations. Broad refinement guidance improves *Shared Core* coverage but leaves a substantial residual gap. These results show that implementing a high-level software task and reproducing the behavioral expectations consistently exhibited by mature implementations remain distinct capabilities. Code, data, and evaluation artifacts are available in an anonymous repository at `https://anonymous.4open.science/r/ShapingBench`.

1 INTRODUCTION

Software rarely remains at the level of its initial requirements. Software evolves as it is used and maintained over time (Lehman, 1980), responding to feedback from users and problems encountered in practice (Van Oordt & Guzman, 2021). Maintenance therefore involves more than preserving original functionality. It includes correction, adaptation, enhancement, and other changes needed to keep a system useful in its environment (ISO/IEC/IEEE, 2022). Over time, these changes accumulate into behavioral decisions that were never part of the original requirements but nevertheless become part of what users expect the software to do. Mature software is therefore shaped not only by its initial specification, but also by the accumulated consequences of real-world use, maintenance, and evolution.

At the same time, coding agents have become increasingly capable of turning explicit intent into working software. Language models can generate programs from natural-language descriptions (Chen et al., 2021), while modern agents can navigate repositories, resolve real software issues (Jimenez et al., 2024), and operate over increasingly long software engineering workflows (Yang et al., 2024; Merrill et al., 2026). Benchmarks have followed this progression, moving from function-level generation (Chen et al., 2021) and programming problems (Austin et al., 2021; Hendrycks et al., 2021) toward repository-level reasoning (Liu et al., 2024; Ding et al., 2023; Li et al., 2024a), real issue resolution (Jimenez et al., 2024), whole-program reconstruction (Yang et al., 2026), and software evolution over releases (Le et al., 2025). Across these settings, the target

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.

behavior is usually provided to the agent through a function description, issue report, reference program, release objective, or similar task input. Existing benchmarks therefore primarily ask whether agents can implement behavior they are told about.

What remains much less understood is how much behavior agents can realize when nobody tells them about it explicitly. Existing evaluations largely test whether agents can implement explicit specifications, not whether they can infer the edge cases, exception handling, compatibility rules, and standards details that mature software acquires in practice. Such expectations are often distributed across tests, bug fixes, and implementation decisions rather than collected in a single specification. Prior software engineering work has studied how requirements and behavioral specifications can be inferred from existing systems and their executions (Liu et al., 1999; Ammons et al., 2002; Charrada, 2010). Recent work on coding agents points to the same difficulty. Recovering implicit requirements is a major source of failure in repository-level tasks (Zhou et al., 2026), while agents perform better when they can probe a reference executable to elicit behavioral specifications (Chen et al., 2026). Comparing independently evolved implementations offers another way to expose mature behaviors that were never written down explicitly and to identify which of them repeatedly emerge across different systems. What remains unclear is how much of this behavior an agent can realize from only a high-level description, without access to a reference implementation or behavioral oracle.

To study this question, we introduce **ShapingBench**, which derives hidden behavioral expectations from multiple mature implementations of the same software domain. Behaviors reproduced by all references form the *Shared Core*, providing the clearest cross-implementation evidence of a recurring expectation. Behaviors reproduced by only a subset form the *Variable Surface*, which characterizes how agent coverage changes as cross-implementation recurrence decreases rather than defining behavior that every implementation should satisfy. Recurrence therefore measures empirical consistency across independently evolved systems, not behavioral importance. This structure allows us to measure not only whether an agent produces functional software, but also how much behavior repeatedly exhibited by mature implementations it realizes without being individually specified in the task.

We instantiate **ShapingBench** across 9 software domains using five mature, independently developed open-source implementations per domain. The resulting 45 references yield 14,537 executable contracts, including 4,754 *Shared Core* and 9,783 *Variable Surface* contracts. Of the Variable Surface, 57.7% recurs in at least two references, showing that it contains substantial recurring behavior rather than only implementation-specific choices.

To evaluate this question, we test 4 LLMs across 3 agent frameworks, covering 10 model–framework combinations and 90 development trajectories. Each agent receives only a high-level task description and executable interface, while the behavioral contracts remain hidden. No evaluated configuration fully realizes the *Shared Core*, with coverage ranging from 62.1% to 88.2% even though every such behavior is reproduced by all five references. Coverage decreases further as recurrence falls, reaching 12.4%–21.3% for behaviors observed in only one reference. Additional development improves coverage but does not eliminate the *Shared Core* gap. These results expose a gap between implementing a high-level task and reproducing behavioral expectations exhibited by mature software.

Our main contributions are:

- We formulate behavioral completeness beyond explicit requirements as an evaluation setting in which mature behavioral expectations remain hidden from the implementation task.
- We introduce **ShapingBench**, comprising 14,537 executable contracts recovered from 45 mature implementations across 9 software domains and organized by cross-implementation recurrence.
- We show that no evaluated configuration fully realizes the *Shared Core*, while behavioral coverage decreases consistently as cross-implementation recurrence falls.

2 Related Work

Evaluation of code generation and software engineering agents. Evaluation of code-generating models has progressively moved from isolated function synthesis toward larger and more realistic

software artifacts. Early work focused on short programs and programming problems (Chen et al., 2021; Austin et al., 2021; Hendrycks et al., 2021; Li et al., 2022), followed by multilingual generation, library-oriented programming, and more rigorous execution-based evaluation (Cassano et al., 2023; Lai et al., 2023; Liu et al., 2023; Zhuo et al., 2025). More recent work evaluates class-level and repository-level generation, where models must account for dependencies, cross-file context, and surrounding implementation structure (Du et al., 2023; Li et al., 2024b; Le Hai et al., 2025; Liu et al., 2024; Ding et al., 2023). Evaluation has further expanded to real issue resolution, autonomous software engineering, terminal-based development, and reconstruction of complete programs (Jimenez et al., 2024; Yang et al., 2024; Merrill et al., 2026; Yang et al., 2026). These benchmarks greatly increase the scale and realism of the implementation task, but the target behavior is still supplied through an issue, requirement, documentation, reference program, or another explicit source. Even whole-program reconstruction provides a reference executable whose behavior can be queried (Yang et al., 2026).

Software evolution and iterative development. Software accumulates behavior through continued maintenance and evolution, including enhancement, adaptation, bug fixing, and changes driven by use and environmental requirements (Lehman, 1980; ISO/IEC/IEEE, 2022; Van Oordt & Guzman, 2021). Repository histories have been mined to identify recurring change relationships and guide future modifications (Zimmermann et al., 2005), while longitudinal studies have examined how production code and tests evolve together (Zaidman et al., 2011). Recent evaluations of coding agents have also moved toward longer development horizons, including release-level software changes, repeated self-improvement, and adaptation from feedback across multiple rounds (Le et al., 2025; Zhang et al., 2026; You et al., 2024). This line of work asks whether agents can carry out a known evolution objective over a longer horizon.

Mining implicit behavioral specifications. A long line of software engineering research has sought to recover specifications from existing software when complete formal requirements are unavailable. Dynamic invariant detection infers recurring properties from executions (Ernst et al., 2001), while source- and trace-based methods recover likely correctness rules, API protocols, and temporal usage constraints from observed program behavior (Engler et al., 2001; Ammons et al., 2002; Weimer & Necula, 2005; Yang et al., 2006; Pradel & Gross, 2009). Differential testing similarly exploits agreement and disagreement among independent implementations when a complete external oracle is unavailable (McKeeman, 1998; Yang et al., 2011). Recent coding-agent work also shows that recovering implicit requirements remains difficult (Zhou et al., 2026), and that explicitly probing a reference executable can improve the behavioral specification available during synthesis (Chen et al., 2026). These approaches recover or expose missing specification by giving the analysis access to an existing system, execution trace, repository, or reference oracle.

Taken together, prior work shows how software behavior evolves over time, how behavioral properties can be inferred from existing systems, and how coding agents perform when the target behavior is explicitly available. **ShapingBench** introduces a different evaluation setting in which behavior accumulated across independently evolved implementations becomes the hidden target. This makes it possible to measure behavioral completeness beyond explicit requirements and relate agent performance to how broadly each behavior recurs across the software ecosystem.

3 SHAPINGBENCH

3.1 DESIGN

ShapingBench evaluates how much behavior exhibited by mature software a coding agent can realize without being given those behaviors explicitly. As illustrated in Figure 1, the method first recovers and validates externally observable behavior from independently evolved implementations, measures how broadly each behavior recurs across them, and then uses the resulting contracts as a hidden evaluation surface for a newly generated implementation.

Behavioral Contracts and Cross-Implementation Support. We define a software domain d as a class of independently developed systems that provide the same broad user-facing functionality and whose externally observable behavior can be compared under a common behavioral abstraction.

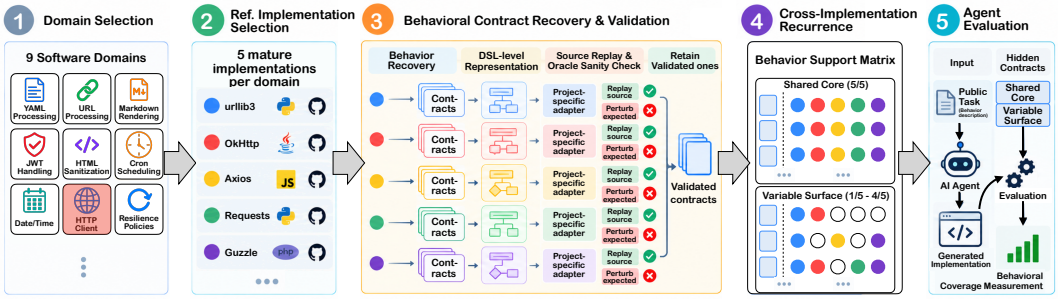


Figure 1: Overview of **ShapingBench**. For each software domain, we select five mature, independently developed reference implementations and recover externally observable behavior as executable contracts. Contracts are normalized into a domain-level representation, replayed on their source implementations, and subjected to an oracle sanity check before being retained. The validated contracts are then replayed across all references to measure cross-implementation recurrence, defining the *Shared Core* (5/5) and *Variable Surface* (1/5–4/5). These contracts remain hidden while a coding agent develops a new implementation from the public task, after which behavioral coverage is measured.

To characterize behavior within a domain, **ShapingBench** uses multiple mature implementations as independent reference points. Reference implementations are selected from actively maintained open-source projects with a long development history, as evidenced by their release records over time. We denote the reference set and its size as

$$\mathcal{P}_d = \{p_1, \ldots, p_{n_d}\}, \qquad n_d = |\mathcal{P}_d|.$$

The choice of n_d determines how finely cross-implementation recurrence can be measured. More references provide finer recurrence levels but increase the cost of cross-implementation evaluation. The implementations in $\mathcal{P}_d$ may differ beyond their shared domain-level functionality and are treated as independently evolved realizations of the same software domain.

Behavioral evidence is recovered from the evolution of each reference implementation, restricted to behavior that can be externally observed and verified. Each recovered behavior is represented as an executable contract $c = (x_c, y_c, O_c)$, where x_c specifies an interaction with the software, y_c is the expected observable outcome, and O_c determines whether an observed outcome satisfies that expectation. Let $\mathcal{R}_d$ denote the recovered contracts for domain d, let p_c^{src} denote the source implementation from which contract c was recovered, and let $A(p, x)$ denote the observable outcome obtained by executing interaction x on implementation p.

Recovered contracts are validated before cross-implementation comparison. Each contract is replayed on the latest release of its source implementation, and the oracle is sanity-checked by perturbing the expected observation and requiring rejection of the perturbed expectation. Specifically,

$$\mathrm{Pos}(c) = O_c(A(p_c^{\mathrm{src}}, x_c), y_c)\,,$$
$$\mathrm{Sanity}(c) = \mathbb{1}[O_c(A(p_c^{\mathrm{src}}, x_c), \tilde{y}_c) = 0]\,,$$

where $\tilde{y}_c$ is a perturbed version of y_c. We retain

$$\mathcal{Q}_d = \{c \in \mathcal{R}_d : \mathrm{Pos}(c) = 1 \wedge \mathrm{Sanity}(c) = 1\}.$$

Thus, $\mathrm{Pos}(c)$ verifies source replay, while $\mathrm{Sanity}(c)$ verifies sensitivity to changes in the scored observation. This is an oracle sanity check rather than a corpus-wide mutation-analysis guarantee. Beyond the source-replay and oracle-sanity checks used during contract construction, we conducted a behavioral-mutant validity audit across the full benchmark. Among 22,587 valid mutants that changed a contract-required observable behavior, the unchanged scoring oracles rejected 21,861 (96.8%). The audit covered 13,330 of 14,537 scoring rows with at least one valid mutant; remaining uncovered rows and surviving mutants are analyzed in Appendix G.

Each validated contract is then replayed across the reference implementations to determine how broadly its behavior recurs. For $c \in \mathcal{Q}_d$ and $p \in \mathcal{P}_d$, we define

$$V(c, p) = O_c(A(p, x_c), y_c)\,, \qquad s_d(c) = \sum_{p \in \mathcal{P}_d} V(c, p),$$

and

$$\mathcal{B}_{d,k} = \{c \in \mathcal{Q}_d : s_d(c) = k\}, \qquad k \in \{1, \ldots, n_d\}.$$

The value $s_d(c)$ gives the number of independent reference implementations that exhibit the behavior represented by c, and $\mathcal{B}_{d,k}$ contains behaviors observed in exactly k references. By construction, mean reference coverage within $\mathcal{B}_{d,k}$ is k/n_d, so reference coverage within a recurrence stratum is a construction identity rather than an independent performance baseline. These recurrence strata define two broader regions of the behavioral surface:

$$\mathcal{C}_d = \mathcal{B}_{d,n_d}, \qquad \mathcal{V}_d = \bigcup_{k=1}^{n_d-1} \mathcal{B}_{d,k}.$$

$\mathcal{C}_d$ is the *Shared Core*, containing behaviors observed in every reference implementation, while $\mathcal{V}_d$ is the *Variable Surface*, containing behaviors observed in only a subset. Cross-implementation support measures recurrence rather than importance. A smaller value of $s_d(c)$ means only that the behavior is observed in fewer independent reference implementations.

Agent Development and Hidden Evaluation. For each domain d, the agent receives a public task x_d describing the software to build at a high level together with the interface required for executable evaluation. The contracts in $\mathcal{Q}_d$, their expected outcomes, provenance, and cross-implementation support remain hidden. The agent therefore develops an implementation S_d without access to the individual behaviors used for evaluation.

The agent starts from an empty workspace and develops S_d iteratively. The implementation produced in earlier iterations is preserved, while the agent interaction is restarted for each subsequent iteration. Feedback between iterations may identify broad areas for further development but does not reveal individual contracts, expected outcomes, or recurrence levels.

The hidden contracts are executed against the generated implementation using the same contract-specific oracles applied to the reference implementations. For $c \in \mathcal{Q}_d$, we define

$$E(c, S_d) = O_c(A(S_d, x_c), y_c)\,.$$

Behavioral coverage over any contract subset $Q \subseteq \mathcal{Q}_d$ is

$$\mathrm{Cov}(S_d, Q) = \frac{1}{|Q|} \sum_{c \in Q} E(c, S_d).$$

Coverage at each recurrence level and over the full behavioral surface is

$$C_{d,k}(S_d) = \mathrm{Cov}(S_d, \mathcal{B}_{d,k}), \qquad C_{d,\mathrm{full}}(S_d) = \mathrm{Cov}(S_d, \mathcal{Q}_d).$$

$C_{d,k}$ measures coverage at recurrence level k, with C_{d,n_d} corresponding to *Shared Core* coverage. $C_{d,\mathrm{full}}$ measures coverage over the complete validated surface $\mathcal{Q}_d = \mathcal{C}_d \cup \mathcal{V}_d$. Because the *Variable Surface* contains behaviors observed in only a subset of references, full-surface coverage is a descriptive measure over the observed behavioral landscape rather than a target whose natural optimum is necessarily 100%. Results aggregated across domains are macro-averaged so that domains with larger contract inventories do not dominate the reported values.

The same hidden evaluation is applied to intermediate implementation snapshots to measure how coverage changes with agentic effort. Let $S_d^{(t)}$ denote the implementation after t cumulative agentic steps. Its full-surface coverage is

$$C_{d,\mathrm{full}}^{(t)} = \mathrm{Cov}(S_d^{(t)}, \mathcal{Q}_d).$$

The resulting trajectory measures how behavioral coverage develops as the agent continues to work and whether additional agentic effort continues to realize previously missing behavior.

3.2 Instantiation

We instantiate **ShapingBench** across nine software domains with five reference implementations per domain, yielding 45 open-source implementations in total. Domains are selected when independently developed systems provide comparable user-facing functionality under a common behavioral

Table 1: Composition of **ShapingBench** by cross-implementation recurrence. *Shared Core* denotes 5/5 behaviors and *Variable Surface* denotes 1/5–4/5 behaviors.

Domain / Implementations	***Shared Core***	***Variable Surface***				**Total**
	5/5	**4/5**	**3/5**	**2/5**	**1/5**	
YAML Parsing and Emission PyYAML, eemeli/yaml, go-yaml, js-yaml, ruamel.yaml	1,881	373	258	370	838	3,720
URL and IRI Processing whatwg-url, rust-url, yarl, ada, curl	330	899	582	37	324	2,172
Markdown Parsing and Rendering markdown-it, pulldown-cmark, goldmark, commonmark.js, marked	1,114	288	223	163	371	2,159
JWT Signing and Verification java-jwt, PyJWT, jsonwebtoken, golang-jwt, Nimbus JOSE	517	832	106	30	137	1,622
HTML Sanitization DOMPurify, Java HTML Sanitizer, jsoup, Bleach, sanitize-html	141	190	123	277	831	1,562
Cron and Schedule Expressions cron-parser, cron-utils, croniter, cron-expression, fugit	177	165	183	140	831	1,496
Date, Time, and Time Zones Joda-Time, Noda Time, dateutil, Carbon, date-fns	250	14	77	104	209	654
HTTP Client urllib3, OkHttp, Axios, Requests, Guzzle	211	36	37	65	284	633
Retry, Backoff, and Resilience Policies resilience4j, Failsafe, backoff, exponential-backoff, async-retry	133	7	29	37	313	519
Total	**4,754**	**2,804**	**1,618**	**1,223**	**4,138**	**14,537**

abstraction, and reference implementations are chosen from mature, actively maintained projects with sustained release histories. In the notation of Section 3.1, this gives $n_d = 5$ and five recurrence levels, $k \in \{1, \ldots, 5\}$. The selected domains span parsing and rendering, network and identifier processing, authentication tokens, sanitization, temporal and scheduling semantics, and resilience policies. Appendix A reports the selected reference projects, versions, ecosystems, and release-history evidence used to document project maturity.

Cross-implementation execution. Implementations within the same domain may expose different execution mechanisms, so each contract $c = (x_c, y_c, O_c)$ is expressed in a domain-level behavioral DSL that separates the behavioral requirement from implementation-specific execution. A separate adapter maps the same contract to each reference implementation and returns its observation to the common representation, allowing validated contracts to be replayed across all five references without modification. With $n_d = 5$, $\mathcal{C}_d = \mathcal{B}_{d,5}$ defines the *Shared Core* and $\mathcal{V}_d = \bigcup_{k=1}^{4} \mathcal{B}_{d,k}$ the *Variable Surface*. Appendix F provides an end-to-end worked example of contract recovery, normalization, validation, and cross-implementation replay.

Benchmark composition. Table 1 summarizes the resulting benchmark. **ShapingBench** contains 14,537 executable behavioral contracts, including 4,754 in the *Shared Core* and 9,783 in the *Variable Surface*. The recurrence distribution is broad, with 2,804 contracts appearing in four references, 1,618 in three, 1,223 in two, and 4,138 in one. Of the *Variable Surface*, 5,645 contracts, or 57.7%, recur in at least two independent implementations, showing that a substantial portion reflects behavior that repeatedly emerges without becoming universal. Contract inventories range from 519 contracts for Retry, Backoff, and Resilience Policies to 3,720 for YAML Parsing and Emission, motivating domain-level macro averaging. Figure 2 further organizes the benchmark by behavioral family. Highly recurrent strata contain substantial standards, validation, and representation behavior, while lower-recurrence strata contain a broader mixture of stateful behavior, normalization, resource handling, security, and extensions. Recurrence therefore captures differences in behavioral composition as well as contract count, and the right panel of Figure 2 relates this structure to agent failures analyzed in Section 4.

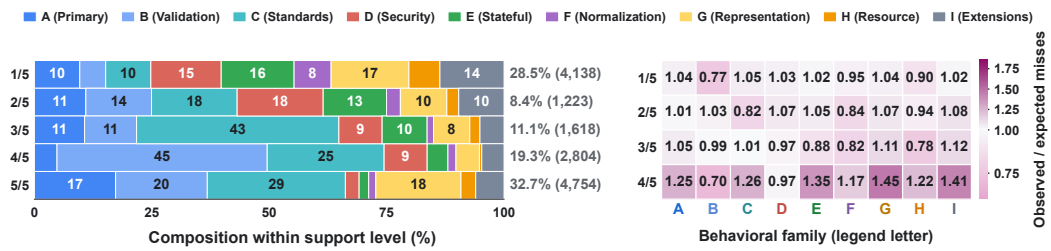


Figure 2: Behavioral composition and agent-miss enrichment across cross-implementation recurrence levels. Left: distribution of behavioral families and benchmark size. Right: observed-to-expected agent-miss ratio for *Variable Surface* contracts, where values above 1 indicate enrichment.

4 Experimental Evaluation

We evaluate 4 LLMs, GPT-5.6-Sol (OpenAI, 2026b), Claude Opus 5 (Anthropic, 2026b), MiniMax M3 (Lai et al., 2026), and GLM 5.3 (GLM-5-Team et al., 2026), on 3 agent frameworks, Codex CLI (OpenAI, 2026a), Claude Code (Anthropic, 2026a), and OpenHands (Wang et al., 2025). The supported model–framework pairings yield 10 configurations and 90 development trajectories across 9 **ShapingBench** domains, with at least 5 agent sessions per trajectory and `high` reasoning effort throughout. All experiments run in isolated execution environments rather than on the local host, where agents may edit files, execute shell commands, install dependencies, build and test implementations, use each framework's native toolchain without interactive approval, and access the network. For each domain, all configurations receive the same public task, while hidden contracts and evaluation outcomes remain inaccessible during development. Appendices B and C provide prompts and execution details, and unless otherwise stated, results are macro-averaged across domains.

4.1 Main Results

We first ask whether behavior that recurs across more mature implementations is also more likely to be realized by a coding agent from the same high-level task. The strongest case is the *Shared Core*, whose behaviors are reproduced by all five reference implementations. Yet no evaluated model–framework configuration reaches complete *Shared Core* coverage: final coverage ranges from 62.1%

Table 2: Final behavioral coverage by cross-implementation recurrence. Values are 9-domain macro averages within each recurrence stratum. Overall reports domain-macro-averaged coverage over the full behavioral surface, and Mean rows average configurations within each framework.

Framework	Model	*Shared Core*	*Variable Surface*				Overall
		5/5	4/5	3/5	2/5	1/5	All
Codex	GPT-5.6-Sol	88.2	79.4	62.8	38.4	14.7	56.8
	MiniMax M3	62.1	45.4	40.0	22.6	13.1	36.6
	GLM 5.3	66.8	63.2	50.6	24.0	12.4	46.6
	Mean	**72.4**	**62.7**	**51.1**	**28.3**	**13.4**	**46.7**
Claude Code	Claude Opus 5	81.4	68.4	66.4	42.5	21.3	57.7
	MiniMax M3	69.7	50.6	43.9	29.2	13.8	42.2
	GLM 5.3	78.0	66.0	59.3	30.5	15.0	51.8
	Mean	**76.4**	**61.7**	**56.5**	**34.1**	**16.7**	**50.6**
OpenHands	GPT-5.6-Sol	78.2	66.7	56.9	30.5	17.2	52.4
	Claude Opus 5	81.0	66.4	63.2	33.7	14.5	52.4
	MiniMax M3	68.0	54.1	44.9	25.7	14.0	42.0
	GLM 5.3	78.0	63.0	53.4	31.7	16.0	51.0
	Mean	**76.3**	**62.6**	**54.6**	**30.4**	**15.4**	**49.5**

to 88.2%. Thus, even behaviors that recur across every mature reference are not fully realized from the high-level task alone.

Beyond this common core, Table 2 shows a consistent monotonic pattern across all 10 model–framework configurations: coverage decreases as recurrence falls from 5/5 to 1/5. Behaviors observed in only one reference achieve only 12.4%–21.3% coverage. The recurrence gradient persists across different models and agent frameworks, indicating that cross-implementation recurrence provides a stable signal of behavioral realizability rather than an artifact of a particular configuration. **ShapingBench** therefore exposes two forms of incompleteness at once: agents fail to fully realize even universally recurring behavior, while increasingly missing additional behavior as cross-implementation recurrence decreases.

4.2 In-Depth Analysis

Behavioral coverage under additional agentic effort. We next examine whether the remaining gap can be closed simply by allowing agents to continue development. Figure 3(a)–(c) tracks full-surface coverage over cumulative agentic steps using the same hidden evaluation on intermediate snapshots. Per-domain trajectory breakdowns are provided in Appendix D. Additional development generally improves coverage, but later checkpoints yield progressively smaller gains, and substantially greater effort does not consistently produce higher final coverage across models, frameworks, or domains. Continued interaction therefore improves behavioral coverage without reliably closing the remaining gap.

Does guidance close the *Shared Core* gap? The *Shared Core* provides a stricter view because its behaviors are reproduced by every reference implementation. Figure 3(d) distinguishes checkpoints before and after broad refinement hints become available without revealing individual contracts or expected outcomes. Across the 82 trajectories that received such hints, *Shared Core* coverage increases from 67.0% at the final pre-hint checkpoint to 72.3% at the final checkpoint; the matched eight-domain cohort shows a similar increase from 72.6% to 78.5%. Because additional development and hint exposure occur together, these comparisons are descriptive rather than causal. Broad guidance improves coverage but does not eliminate the residual *Shared Core* gap.

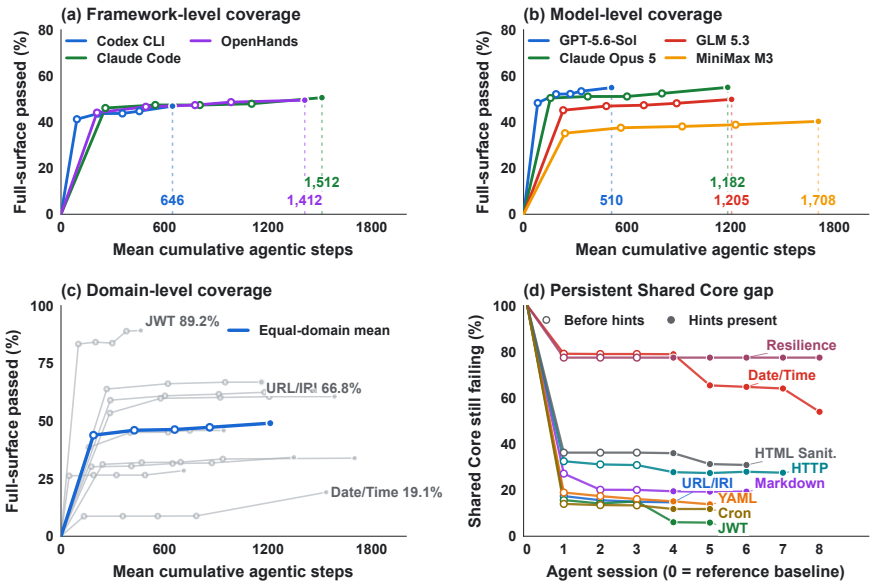


Figure 3: Behavioral coverage over continued agent development. (a)–(c) Full-surface coverage over cumulative agentic steps, aggregated by framework, model, and domain. Open markers denote intermediate checkpoints and filled markers denote final checkpoints. (d) Normalized remaining *Shared Core* failures across agent sessions, with session 0 as the reference baseline. Open markers indicate sessions before broad hints and filled markers indicate sessions with hints available.

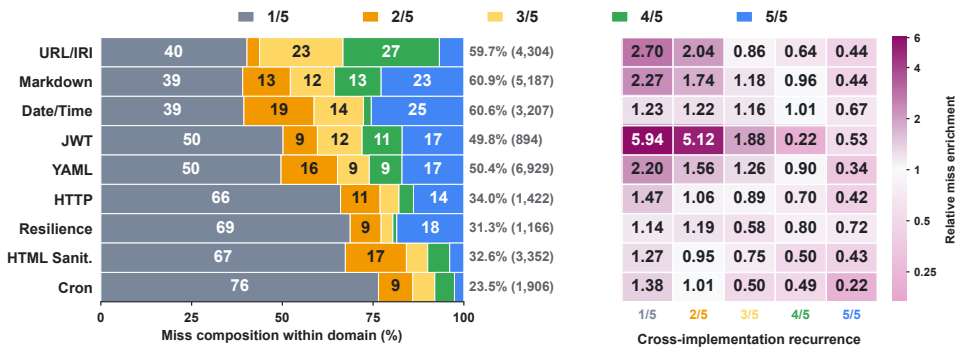


Figure 4: Agent misses by cross-implementation recurrence. Left: miss composition within each domain. Right: relative miss enrichment after normalizing by the prevalence of each recurrence level within the domain; values above 1 indicate overrepresentation among misses.

Which recurrence levels account for the remaining failures? Figure 4 examines residual failures from two complementary views. The left panel shows how misses are distributed across recurrence levels within each domain, while the right panel normalizes this distribution by how prevalent each recurrence level is in that domain. We define relative miss enrichment as

$$R_{d,k} = \frac{M_{d,k}/M_d}{|\mathcal{B}_{d,k}|/|\mathcal{Q}_d|},$$

where $M_{d,k}$ is the number of misses on contracts in $\mathcal{B}_{d,k}$ and M_d is the total number of misses in domain d. Values above 1 indicate that a recurrence level is overrepresented among failures relative to its prevalence in the domain.

Failures are not confined to isolated 1/5 behavior, and every domain retains misses on behaviors reproduced by multiple references. After normalizing for domain composition, low-recurrence behaviors are generally overrepresented among failures, while 4/5 and 5/5 behaviors are underrepresented. The recurrence gradient therefore cannot be explained simply by how many low-recurrence contracts each domain contains, although its strength still varies across domains. Model- and domain-specific breakdowns are provided in Appendix E.

Behavioral difficulty beyond recurrence. Cross-implementation recurrence captures only one dimension of difficulty. The right panel of Figure 2 compares behavioral families within the same recurrence level and shows that miss enrichment remains uneven even when recurrence is fixed. At 4/5 recurrence, representation, stateful behavior, standards-related behavior, and extensions are more strongly represented among misses than validation behavior. Contracts with similar recurrence can therefore differ substantially in realizability.

Together, these analyses reveal a layered structure of behavioral incompleteness. Recurrence defines a strong global gradient, while domain characteristics and behavioral family explain additional variation. Continued development and broad guidance improve coverage without eliminating the gap, including for behavior consistently reproduced across mature software.

5 Conclusion

We introduced **ShapingBench**, a benchmark for evaluating behavioral completeness beyond stated requirements. Across 9 software domains, 45 mature implementations, and 14,537 executable contracts, no evaluated configuration fully realizes the *Shared Core* of behaviors reproduced by all five references. Beyond this common core, cross-implementation recurrence provides a consistent gradient of behavioral realizability, while continued development and broad guidance improve coverage without eliminating the residual *Shared Core* gap. These results show that implementing a high-level task and reproducing the behavioral expectations exhibited by mature software remain meaningfully different capabilities.

AI USE STATEMENT

Generative AI tools were used to assist with language polishing, literature retrieval and discovery, and research execution including software development and analysis. All AI-assisted outputs and retrieved references were reviewed and verified by the authors, and all experimental results were produced and validated using the procedures described in the paper. The authors remain responsible for all scientific claims, analyses, and conclusions.

REPRODUCIBILITY STATEMENT

We provide the benchmark artifacts and code at `https://anonymous.4open.science/r/ShapingBench`. Section 3 describes benchmark construction and evaluation, while the appendices document the reference implementations, public agent prompts, execution configurations, per-domain analyses, an end-to-end contract construction example, and the behavioral-mutant validity audit. The released artifacts include the executable contracts, evaluation code, and supporting materials needed to reproduce the reported analyses.

REFERENCES

Glenn Ammons, Rastislav Bodík, and James R Larus. Mining specifications. *ACM Sigplan Notices*, 37(1):4–16, 2002.

Anthropic. Claude code, 2026a. URL `https://docs.anthropic.com/en/docs/claude-code`.

Anthropic. Claude opus 5 system card, 2026b. URL `https://www-cdn.anthropic.com/c5fbac3f0b1280a933ebd26d3cb8bb9f5bdeaf48/Claude%20Opus%205%20System%20Card.pdf`.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. Multipl-e: A scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*, 49(7):3675–3691, 2023.

Eya Ben Charrada. Updating requirements from tests during maintenance and evolution. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 337–340, 2010.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Yihao Chen, Shi Chang, Feng Lin, Khaled Chawa, Boyuan Chen, Shaowei Wang, and Ahmed E. Hassan. Specfirst: Behavioral specification elicitation as a first-class step in agent-based program synthesis from scratch, 2026. URL `https://arxiv.org/abs/2607.27167`.

Yangruibo Ding, Zijian Wang, Wasi Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, et al. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. *Advances in Neural Information Processing Systems*, 36:46701–46723, 2023.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. *arXiv preprint arXiv:2308.01861*, 2023.

Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. *ACM SIGOPS Operating Systems Review*, 35(5):57–72, 2001.

Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE transactions on software engineering*, 27(2):99–123, 2001.

GLM-5-Team, :, Aohan Zeng, Xin Lv, Zhenyu Hou, Zhengxiao Du, Qinkai Zheng, Bin Chen, Da Yin, Chendi Ge, Chenghua Huang, Chengxing Xie, Chenzheng Zhu, Congfeng Yin, Cunxiang Wang, Gengzheng Pan, Hao Zeng, Haoke Zhang, Haoran Wang, Huilong Chen, Jiajie Zhang, Jian Jiao, Jiaqi Guo, Jingsen Wang, Jingzhao Du, Jinzhu Wu, Kedong Wang, Lei Li, Lin Fan, Lucen Zhong, Mingdao Liu, Mingming Zhao, Pengfan Du, Qian Dong, Rui Lu, Shuang-Li, Shulin Cao, Song Liu, Ting Jiang, Xiaodong Chen, Xiaohan Zhang, Xuancheng Huang, Xuezhen Dong, Yabo Xu, Yao Wei, Yifan An, Yilin Niu, Yitong Zhu, Yuanhao Wen, Yukuo Cen, Yushi Bai, Zhongpei Qiao, Zihan Wang, Zikang Wang, Zilin Zhu, Ziqiang Liu, Zixuan Li, Bojie Wang, Bosi Wen, Can Huang, Changpeng Cai, Chao Yu, Chen Li, Chengwei Hu, Chenhui Zhang, Dan Zhang, Daoyan Lin, Dayong Yang, Di Wang, Ding Ai, Erle Zhu, Fangzhou Yi, Feiyu Chen, Guohong Wen, Hailong Sun, Haisha Zhao, Haiyi Hu, Hanchen Zhang, Hanrui Liu, Hanyu Zhang, Hao Peng, Hao Tai, Haobo Zhang, He Liu, Hongwei Wang, Hongxi Yan, Hongyu Ge, Huan Liu, Huanpeng Chu, Jia'ni Zhao, Jiachen Wang, Jiajing Zhao, Jiamin Ren, Jiapeng Wang, Jiaxin Zhang, Jiayi Gui, Jiayue Zhao, Jijie Li, Jing An, Jing Li, Jingwei Yuan, Jinhua Du, Jinxin Liu, Junkai Zhi, Junwen Duan, Kaiyue Zhou, Kangjian Wei, Ke Wang, Keyun Luo, Laiqiang Zhang, Leigang Sha, Liang Xu, Lindong Wu, Lintao Ding, Lu Chen, Minghao Li, Nianyi Lin, Pan Ta, Qiang Zou, Rongjun Song, Ruiqi Yang, Shangqing Tu, Shangtong Yang, Shaoxiang Wu, Shengyan Zhang, Shijie Li, Shuang Li, Shuyi Fan, Wei Qin, Wei Tian, Weining Zhang, Wenbo Yu, Wenjie Liang, Xiang Kuang, Xiangmeng Cheng, Xiangyang Li, Xiaoquan Yan, Xiaowei Hu, Xiaoying Ling, Xing Fan, Xingye Xia, Xinyuan Zhang, Xinze Zhang, Xirui Pan, Xu Zou, Xunkai Zhang, Yadi Liu, Yandong Wu, Yanfu Li, Yidong Wang, Yifan Zhu, Yijun Tan, Yilin Zhou, Yiming Pan, Ying Zhang, Yinpei Su, Yipeng Geng, Yong Yan, Yonglin Tan, Yuean Bi, Yuhan Shen, Yuhao Yang, Yujiang Li, Yunan Liu, Yunqing Wang, Yuntao Li, Yurong Wu, Yutao Zhang, Yuxi Duan, Yuxuan Zhang, Zezhen Liu, Zhengtao Jiang, Zhenhe Yan, Zheyu Zhang, Zhixiang Wei, Zhuo Chen, Zhuoer Feng, Zijun Yao, Ziwei Chai, Ziyuan Wang, Zuzhou Zhang, Bin Xu, Minlie Huang, Hongning Wang, Juanzi Li, Yuxiao Dong, and Jie Tang. Glm-5: from vibe coding to agentic engineering, 2026. URL https://arxiv.org/abs/2602.15763.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*, 2021.

ISO/IEC/IEEE. Iso/iec/ieee international standard - software engineering - software life cycle processes - maintenance. *ISO/IEC/IEEE 14764:2022(E)*, pp. 1–46, 2022. doi: 10.1109/IEEESTD.2022.9690131.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024, pp. 54107–54157, 2024.

Xunhao Lai, Weiqi Xu, Yufeng Yang, Qiaorui Chen, Yang Xu, Lunbin Zeng, Xiaolong Li, Haohai Sun, Haichao Zhu, Vito Zhang, et al. Minimax sparse attention. *arXiv preprint arXiv:2606.13392*, 2026.

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 *International Conference on Machine Learning*, pp. 18319–18345. PMLR, 2023.

Tue Le, Minh VT Thai, Dung Nguyen Manh, Huy Phan Nhat, and Nghi DQ Bui. Swe-evo: Benchmarking coding agents in long-horizon software evolution scenarios. *arXiv preprint arXiv:2512.18470*, 2025.

Nam Le Hai, Dung Manh Nguyen, and Nghi DQ Bui. On the impacts of contexts on repository-level code generation. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 1496–1524, 2025.

M.M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68 (9):1060–1076, 1980. doi: 10.1109/PROC.1980.11805.

Jia Li, Ge Li, Xuanming Zhang, Yihong Dong, and Zhi Jin. Evocodebench: An evolving code generation benchmark aligned with real-world code repositories. *arXiv preprint arXiv:2404.00599*, 2024a.

Jia Li, Ge Li, Yunfei Zhao, Yongmin Li, Huanyu Liu, Hao Zhu, Lecheng Wang, Kaibo Liu, Zheng Fang, Lanshen Wang, et al. Deveval: A manually-annotated code generation benchmark aligned with real-world code repositories. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 3603–3614, 2024b.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in neural information processing systems*, 36:21558–21572, 2023.

Kecheng Liu, Albert Alderson, and Zubair Qureshi. Requirements recovery from legacy systems by analysing and modelling behaviour. In *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*, pp. 3–12. IEEE, 1999.

Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems. In *International Conference on Learning Representations*, volume 2024, pp. 47832–47850, 2024.

William M McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

Mike Merrill, Alexander Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *International Conference on Learning Representations*, volume 2026, pp. 40903–40986, 2026.

OpenAI. Codex CLI, 2026a. URL https://github.com/openai/codex.

OpenAI. GPT-5.6 system card, 2026b. URL https://deploymentsafety.openai.com/gpt-5-6.

Michael Pradel and Thomas R Gross. Automatic generation of object usage specifications from large method traces. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pp. 371–382. IEEE, 2009.

Simon Van Oordt and Emitza Guzman. On the role of user feedback in software evolution: A practitioners' perspective. In Ana Moreira, Kurt Schneider, Michael Vierhauser, and Jane Cleland-Huang (eds.), *2021 IEEE - 29th IEEE International Requirements Engineering Conference (RE)*, Proceedings of the IEEE International Conference on Requirements Engineering, pp. 221–232, United States, November 2021. IEEE Computer Society. ISBN 9781665428576. doi: 10.1109/RE51729.2021.00027. Publisher Copyright: © 2021 IEEE.; 29th IEEE International Requirements Engineering Conference, RE 2021 ; Conference date: 20-09-2021 Through 24-09-2021.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Daniel Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for ai software developers as generalist agents. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 65882–65919, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/a4b6ad6b48850c0c331d1259fc66a69c-Paper-Conference.pdf.

Westley Weimer and George C Necula. Mining temporal specifications for error detection. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 461–476. Springer, 2005.

Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: mining temporal api rules from imperfect traces. In *Proceedings of the 28th international conference on Software engineering*, pp. 282–291, 2006.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

John Yang, Kilian Lieret, Jeffrey Ma, Parth Thakkar, Dmitrii Pedchenko, Sten Sootla, Emily McMilin, Pengcheng Yin, Rui Hou, Gabriel Synnaeve, et al. Programbench: Can language models rebuild programs from scratch? *arXiv preprint arXiv:2605.03546*, 2026.

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pp. 283–294, 2011.

Jiaxuan You, Mingjie Liu, Shrimai Prabhumoye, Mostofa Patwary, Mohammad Shoeybi, and Bryan Catanzaro. Llm-evolve: Evaluation for llm's evolving capability on benchmarks. In *Proceedings of the 2024 conference on empirical methods in natural language processing*, pp. 16937–16942, 2024.

Andy Zaidman, Bart Van Rompaey, Arie Van Deursen, and Serge Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 16(3):325–364, 2011.

Wentao Zhang, Jianfeng Wang, Liheng Liang, Yilei Zhao, HaiBin Wen, and Zhe Zhao. Evocodebench: A human-performance benchmark for self-evolving llm-driven coding systems. *arXiv preprint arXiv:2602.10171*, 2026.

Xin Zhou, Chun Yong Chong, Kisub Kim, Yun Peng, Rui Shu, Zihan Wu, Xu Han, Guowen Yuan, Zeyang Zhuang, Jounghoon Kim, Jeongjin Ju, Seongmin Ju, Taein Yoon, and David Lo. A unified issue resolution benchmark for requirement clarification, planning, and code generation for coding agents, 2026. URL `https://arxiv.org/abs/2608.09072`.

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 *International Conference on Learning Representations*, volume 2025, pp. 66602–66656, 2025.

Thomas Zimmermann, Andreas Zeller, Peter Weissgerber, and Stephan Diehl. Mining version histories to guide software changes. *IEEE Transactions on software engineering*, 31(6):429–445, 2005.

A Reference Implementations and Maturity

ShapingBench uses five independently developed reference implementations for each software domain. Reference projects were selected from maintained open-source software with stable tagged releases, release or changelog histories, behavioral tests or documentation, and versions that could be locally built or replayed. The selection also favored independent repositories and diversity in languages, package ecosystems, and public APIs so that cross-implementation recurrence would not simply reflect a shared implementation lineage. Popularity metrics such as stars or downloads were not used as threshold criteria.

Table 3 groups the 45 references by domain and reports the selected version together with the release-history evidence available during benchmark construction. The evidence varies by ecosystem, so we use it to document sustained project evolution rather than to construct a cross-project maturity score.

Table 3: Reference implementations used in ShapingBench and the release-history evidence used to establish project maturity.

Domain	Project	Language / Ecosystem	Version	Release History
YAML Parsing and Emission	PyYAML	Python / PyPI	6.0.3	40 PyPI releases; 39 matched Git tags
	eemeli/yaml	JavaScript / npm	2.9.0	103 npm versions; 94 matched Git tags; records since 2011
	go-yaml	Go / modules	v3.0.1	16 Git tags; records since 2018
	js-yaml	JavaScript / npm	5.4.1	90 releases with recoverable source; records since 2011
	ruamel.yaml	Python / PyPI	0.19.1	273 PyPI releases; 234 with recoverable source
URL and IRI Processing	whatwg-url	JavaScript / npm	17.1.0	57 releases with notes; 72 npm versions
	rust-url	Rust / crates.io	2.5.8	89 crates.io versions; 9 GitHub releases with notes
	yarl	Python / PyPI	1.24.5	138 PyPI versions; 119 changelog versions; 91 GitHub releases
	ada	C++ / GitHub	4.0.0	61 GitHub releases; 62 Git tags
	curl	C / releases	8.21.0	135 URL-API release items; 196 release tags inspected
Markdown Parsing and Rendering	markdown-it	JavaScript / npm	15.0.1	85 npm versions with matching Git tags
	pulldown-cmark	Rust / crates.io	0.13.4	50 crates.io versions; 49 Git tags
	goldmark	Go / modules	2.0.0	118 Git tags; 116 SemVer release points
	commonmark.js	JavaScript / npm	0.31.2	29 npm versions with matching release tags
	marked	JavaScript / npm	18.0.11	222 npm versions; 218 matched release tags
JWT Signing and Verification	java-jwt	Java / Maven	4.6.0	62 Git tags; 61 Maven versions
	PyJWT	Python / PyPI	2.13.0	55 PyPI releases; 49 Git tags
	jsonwebtoken	JavaScript / npm	9.0.3	82 npm releases; 75 Git tags
	golang-jwt	Go / modules	v5.3.1	37 Git tags
	Nimbus JOSE	Java / Maven	10.9.1	318 Git tags

Domain	Project	Language / Ecosystem	Version	Release History
HTML Sanitization	DOMPurify	JavaScript / npm	3.4.14	151 npm versions; 148 GitHub releases
	Java HTML Sanitizer	Java / Maven	20260313.1	42 Maven versions; 28 Git tags
	jsoup	Java / Maven	1.23.2	63 release points; 56 Maven versions
	Bleach	Python / PyPI	6.4.0	61 PyPI releases; 64 Git tags
	sanitize-html	JavaScript / npm	2.17.7	126 npm versions; 55 Git tags
Cron and Schedule Expressions	cron-parser	JavaScript / npm	5.10.0	92 release-note versions; contracts recovered from 54 releases
	cron-utils	Java / Maven	9.2.1	42 GitHub releases; records since 2015
	croniter	Python / PyPI	6.2.4	106 PyPI releases; 95 Git tags; records since 2010
	cron-expression	PHP / Packagist	3.6.0	29 stable versions; 20 changelog releases; records since 2020
	fugit	Ruby / RubyGems	1.13.0	61 changelog releases, Git tags, and stable RubyGems versions
Date, Time, and Time Zones	Joda-Time	Java / Maven	2.14.3	558 release-note items; 66 Maven releases
	Noda Time	C# / NuGet	3.3.3	75 release sections
	dateutil	Python / PyPI	2.9.0.post0	32 release sections
	Carbon	PHP / Packagist	3.13.2	270 Git tags; 100 cached GitHub releases
	date-fns + @date-fns/tz	JavaScript / npm	4.4.0 + 1.5.0	213 Git tags; 44 core and 11 timezone changelog releases
HTTP Client	urllib3	Python / PyPI	2.7.0	110 releases; records since 2008
	OkHttp	Kotlin/Java / Maven	5.5.0	140 releases scanned; contracts recovered from 84; records since 2013
	Axios	JavaScript / npm	1.19.0	145 npm versions; 107 releases with notes; records since 2014
	Requests	Python / PyPI	2.32.5	162 release-note versions
	Guzzle	PHP / Packagist	8.1.0	109 release-note versions
Retry, Backoff, and Resilience Policies	resilience4j	Java / Maven	2.4.0	36 GitHub releases; 32 Maven versions; records since 2016
	Failsafe	Java / Maven	3.3.2	54 changelog releases; 46 Maven versions
	backoff	Python / PyPI	2.2.1	37 PyPI releases; 27 GitHub releases; records since 2014
	exponential-backoff	JavaScript / npm	3.1.3	18 npm versions; records since 2018
	async-retry	JavaScript / npm	1.3.3	18 GitHub releases; 18 npm versions; records since 2016

Table 4: Onset of broad refinement prompts across the primary configurations.

Domain	Refinement Prompt Onset
HTTP Client	Session 4 in 10/10 configurations
Date/Time	Session 4 in 10/10 configurations
URL/IRI	Session 4 in 10/10 configurations
HTML Sanitization	Session 4 in 9/10 configurations
Markdown	Session 4 in 9/10 configurations
YAML	Session 4 in 9/10 configurations
JWT	Session 4 in 10/10 configurations
Cron	Session 4 in 10/10 configurations
Resilience	Session 4 in 1/10; Session 6 in 1/10

B Public Tasks and Agent Prompts

For each domain, the agent received a public implementation task describing the software to build and the interface required for executable evaluation. Hidden behavioral contracts, expected outcomes, cross-implementation recurrence, and evaluator traces were not included in the public task. Source-code changes were preserved across sessions, while each subsequent agent interaction started with a fresh conversational context. Later sessions initially requested further development in generic terms, and broad failure-area hints were introduced in most trajectories after several sessions.

Representative prompt texts used in the experiments are reproduced below. For each domain, we show one initial prompt and one broad refinement prompt when such guidance was used. Minor wording variants across model–framework configurations are omitted because they specify the same task and evaluation boundary.

B.1 HTTP CLIENT

Initial prompt.

```
You are working in an empty independent workspace.

Implement a production-quality, dependency-light Python HTTP client library named `solhttp`.

Create whatever files are appropriate, but expose this public API:

- `from solhttp import Client, request, get, post, put, patch, delete, head, options`
- `Client()` creates a reusable client with connection pooling where possible.
- `Client.request(method, url, **kwargs)` performs an HTTP request.
- Top-level `request`, `get`, `post`, `put`, `patch`, `delete`, `head`, and `options`
    functions should work without explicitly constructing a client.
- Responses expose:
  - `status_code`
  - `headers`
  - `content`
  - `text`
  - `url`
  - `ok`
  - `history`
  - `json(**kwargs)`
  - `iter_bytes(chunk_size=1)`
  - `close()`
  - `raise_for_status()`
- Support common request options:
  - `params`
  - `headers`
  - `data`
  - `files`
  - `json`
  - `timeout`
  - `follow_redirects`
  - `stream`
  - `verify`
  - `cert`
  - `proxy`
- Define a useful exception hierarchy including:
  - `SolHTTPError`
  - `RequestError`
  - `InvalidURL`
  - `ConnectionError`
  - `Timeout`
  - `ProtocolError`
  - `HTTPStatusError`
  - `TooManyRedirects`
  - `TimeoutConfig`

Aim for mature behavior across common HTTP client expectations: request construction, URL
    normalization, query parameters, form bodies, multipart file uploads, JSON bodies,
    redirects, cookies, response streaming, decompression, connection reuse, retries where
    appropriate, timeout handling, TLS verification, proxies, error classification, and
    stable response metadata.

Add tests and documentation if useful. Do not ask questions. Implement, test locally, and
    leave the final implementation in this workspace.
```

Broad refinement prompt.

```
You are continuing work on an existing Python HTTP client implementation in this workspace.

This is iteration 4, starting from the iteration 3 implementation. Improve the
    implementation using your own engineering judgment. You may use the following broad
    failure-area hints, but you must not assume access to any hidden benchmark or contract
    corpus.

Focus areas that mature HTTP clients commonly get wrong:

- Timeout and network failures should be consistently wrapped in the library's own exception
     hierarchy, without constructor/type mismatches, and without retrying operations that
    failed because of timeout.
- Redirect handling should preserve or rewrite methods and bodies according to HTTP
    semantics, especially 303 versus 307/308, including methods beyond GET and POST.
- Request bodies should work for methods such as DELETE and OPTIONS, including raw content,
    form data, and streamed/chunked content.
- Chunked request framing should emit valid request boundaries on keep-alive, avoid
    duplicating Transfer-Encoding headers, and produce deterministic chunk framing.
- Form encoding should handle explicit charsets and encode spaces as plus signs where
    appropriate.
- Multipart form uploads should accept common tuple shapes, preserve duplicate field names,
    support plain fields, support non-ASCII filenames, use browser-compatible filename
    header escaping, and emit Content-Length when the multipart body length is known.
- Repeated response headers such as Set-Cookie should remain observable as repeated header
    values, not collapse into a lossy single-value representation.
- Incomplete Content-Length bodies and malformed chunked responses should raise the client
    protocol/error type that callers can catch.
- Proxy support should correctly handle HTTP CONNECT for HTTPS targets, IPv6 proxy hosts,
    trailing-dot hostnames, and separation between proxy headers and origin headers.
- TLS support should cover client certificates and certificate verification for IP address
    subjectAltName cases, including IPv6 bracket handling.
- Optional content codings should either decode correctly or fail/degrade explicitly and
    predictably when the decoder is unavailable.

Preserve the public package name and API style already present in the workspace. Keep the
    implementation dependency-light unless you have a strong reason otherwise. Improve
    correctness, edge-case handling, ergonomics, tests, and documentation as you see fit.

Important constraints:
- Do not remove existing working features.
- Keep the project runnable from this workspace.
- Run an appropriate test suite before finishing.
- Summarize what you changed and what tests passed.
```

B.2 DATE, TIME, AND TIME ZONES

Initial prompt.

```
Implement a compact standalone date/time/timezone library.

Create an executable `./dtlib`. It must read one JSON object from stdin and
write one JSON object to stdout. The input has a `replay` operation name and
optional `params`. Semantic errors should be JSON such as `{"throws":"parse"}`.

Do not inspect benchmark files, evaluator outputs, hidden corpora, or previous
answers. Do not hard-code tests. This is the first no-hint implementation.

Implement broad, practical behavior for dates, times, datetimes, instants,
durations, periods, intervals, fixed offsets, IANA zones, DST gaps/overlaps,
epoch conversion, local/zoned conversion, and common parsing/formatting. Be
generous with operation-name aliases and return specific JSON fields like
`value`, `date`, `time`, `datetime`, `instant`, `offset`, `millis`, `seconds`,
`start`, `end`, `id`, `ids`, `canonical_id`, `exists`, and `ambiguous`.

Keep the implementation compact and create a meaningful local self-test. Avoid
large preliminary exploration and avoid enormous single shell commands.
```

Broad refinement prompt.

```
Improve `./dtlib` using broad natural-language failure-area hints.

Do not look for benchmark files or contract corpora. Do not hard-code a test
corpus. Keep the same executable interface and preserve current working
behavior.

The remaining gaps are likely in these broad areas:

- IANA timezone database presence and canonical zone identifiers.
- Zone aliases and fixed-offset zones without silently coercing offsets to UTC.
- Date/time parser behavior for ISO variants, invalid strings, text case,
  fractional seconds, and signed or reduced forms.
- Local date start-of-day behavior around time zones and DST transitions.
- DST gaps, DST overlaps, ambiguous local times, and imaginary/nonexistent local
  times.
- Intervals: parsing, before/after, abutting, overlap, gap, containment, and
  intersection behavior.
- Period parsing and arithmetic as calendar periods rather than fixed
  durations.
- Partial values such as month-day and year-month.
- Output shape: return specific fields like `canonical_id`, `offset`, `date`,
  `time`, `datetime`, `instant`, `millis`, `start`, `end`, `value`, `exists`,
  and `ambiguous` when they are the natural observable result.

Add tests that actually exercise the behavior you add.
```

B.3 URL AND IRI PROCESSING

Initial prompt.

```
You are working in an empty independent workspace.

Implement a production-quality, dependency-light Python URL/IRI parsing and manipulation
     library named `solurl`.

Create whatever files are appropriate, but expose this public API:

- `from solurl import URL, parse_url, can_parse`
- `URL(input: str, base: str | URL | None = None)` parses an absolute URL, or resolves a
     relative reference against `base`.
- Invalid inputs should raise `ValueError`.
- URL instances expose string properties:
  - `href`
  - `origin`
  - `protocol`
  - `username`
  - `password`
  - `host`
  - `hostname`
  - `port`
  - `pathname`
  - `search`
  - `hash`
- The same components are settable where meaningful:
  - `href`
  - `protocol`
  - `username`
  - `password`
  - `host`
  - `hostname`
  - `port`
  - `pathname`
  - `search`
  - `hash`
- `str(url)` should return `url.href`.
- `parse_url(input, base=None)` should return a `URL`.
- `can_parse(input, base=None)` should return a boolean.
- `URL.can_parse(input, base=None)` should also work.

Aim for mature behavior across common modern URL/IRI expectations: relative resolution,
     component extraction, percent encoding, IDNA host handling, IPv4/IPv6 hosts, file URLs,
      non-special schemes, query and fragment handling, path normalization, setter semantics
     , and stable serialization. Add tests and documentation if useful.

Do not ask questions. Implement, test locally, and leave the final implementation in this
     workspace.
```

Broad refinement prompt.

```
You are working in an existing Python URL/IRI library implementation.

Continue improving the implementation quality and standards coverage while preserving the
    public API:

- `from solurl import URL, parse_url, can_parse`
- `URL(input: str, base: str | URL | None = None)`
- string properties: `href`, `origin`, `protocol`, `username`, `password`, `host`, `hostname
    `, `port`, `pathname`, `search`, `hash`
- the same components should remain settable where meaningful
- invalid inputs should raise `ValueError`
- `str(url)` should return `url.href`
- `parse_url(input, base=None)` should return a `URL`
- `can_parse(input, base=None)` and `URL.can_parse(input, base=None)` should return booleans

Focus on broad URL/IRI maturity: parsing and serialization consistency, relative resolution,
     path normalization, edge-case delimiters, percent-encoding rules, IDNA, IPv4/IPv6,
    file URLs, default ports, setters, invalid inputs, and stable roundtrips.

Pay extra attention to path normalization edge cases where empty path segments and slash
    boundaries must remain observable after resolving `.` or `..` segments. Some mature URL
     implementations preserve a trailing slash introduced by a final current-directory
    segment, and preserve double slash path segments instead of collapsing them into one
    slash.

Do not ask questions. Improve the code, expand or adjust tests, run validation locally, and
    leave the improved implementation in this workspace.
```

B.4 HTML SANITIZATION

Initial prompt.

```
Implement a production-quality, dependency-light Python HTML sanitizer library named `solsanitize`.

Public API:

- `from solsanitize import sanitize, is_valid`
- `sanitize(html: str | None, *, config: dict | None = None, policy: dict | str | None = None, base_url: str | None = None) -> str`
- `is_valid(html: str | None, *, config: dict | None = None, policy: dict | str | None = None, base_url: str | None = None) -> bool`

Expected behavior:

- Accept an HTML fragment or document-ish string and return a sanitized HTML fragment string.
- Remove dangerous scripting, event handlers, unsafe URLs, unsafe attributes, and unsafe namespaced markup.
- Preserve common safe formatting, links, images, tables, SVG/MathML-safe fragments, and text where appropriate.
- Support configurable policies for allowed/forbidden tags, allowed/forbidden attributes, URL protocols, relative URL handling, CSS/style handling, and simple canned policies such as formatting, blocks, links, images, tables, and relaxed/basic/simple-text styles.
- Decode and serialize HTML entities consistently enough for mature sanitizer behavior.
- `is_valid` should report whether sanitization would leave the observable fragment unchanged under the selected policy.

You may add tests, documentation, and dependencies if useful, but keep the package practical and easy to run locally. Do not assume access to any external hidden contract corpus.

Fairness constraint: do not inspect benchmark files, evaluator outputs, hidden corpora, previous generated implementations, previous experiment summaries, or any contract corpus in this repository. Work only from this prompt and your general knowledge.
```

Broad refinement prompt.

```
You are continuing work on an existing Python HTML sanitizer library named `solsanitize`.

Keep the public API stable:

- `from solsanitize import sanitize, is_valid`
- `sanitize(html: str | None, *, config: dict | None = None, policy: dict | str | None = None, base_url: str | None = None) -> str`
- `is_valid(html: str | None, *, config: dict | None = None, policy: dict | str | None = None, base_url: str | None = None) -> bool`

Improve the implementation using broad sanitizer knowledge. Do not add hidden-test special cases and do not weaken security.

General hint from the previous black-box evaluation:

- Most remaining gaps are around SVG / XML-ish edge cases: profile-specific preservation of safe SVG pieces, invalid XML-like markup, comments / empty comments around SVG-ish inputs, and how sanitizer output should remain stable without accidentally over-escaping safe recovered text.
- The other remaining gaps are around jsoup-like safelist semantics: URL protocol handling under safelists, attribute removal vs preservation, enforced attributes on links, and relative URL resolution when a base URI is available.
- Pay attention to serializer compatibility: lowercase/normalized attributes where appropriate, deterministic attribute order, implicit element closure, and whether disallowed tags are unwrapped, dropped with content, or escaped.

Add tests for the broad behaviors above, run the test suite, and clean cache/build artifacts. Do not use external sanitizer libraries; keep it dependency-free unless absolutely necessary.
```

B.5 MARKDOWN PARSING AND RENDERING

Initial prompt.

```
You are working in an empty independent workspace.

Implement a production-quality, dependency-light Python Markdown parser/renderer library
    named `solmarkdown`.

Create whatever files are appropriate, but expose this public API:

- `from solmarkdown import render, render_inline`
- `render(markdown: str | None, *, options: dict | None = None) -> str`
- `render_inline(markdown: str | None, *, options: dict | None = None) -> str`

Expected behavior:

- Convert Markdown text to deterministic HTML.
- Support mature CommonMark-style behavior across paragraphs, emphasis, strong emphasis,
    code spans, fenced and indented code blocks, block quotes, ATX and Setext headings,
    thematic breaks, ordered and unordered lists, nested lists, links, images, autolinks,
    raw HTML handling, entities, escaping, hard and soft line breaks, and edge-case parser
    regressions.
- Support practical extension behavior where enabled through `options`, including tables,
    strikethrough, task lists, typographer/smart punctuation, linkify/autolink-style URL
    handling, heading attributes and generated heading IDs, definition lists, footnotes,
    metadata/front-matter style blocks, math-ish spans/blocks, wiki links, XHTML-style
    empty tags, unsafe/raw HTML toggles, and line-break options.
- `render_inline` should render inline Markdown without wrapping it in a paragraph.
- `options` may contain booleans, nested option dictionaries, an `enabled` list, or preset-
    style values. Treat unknown options safely.
- HTML serialization should be stable and conventional enough for mature Markdown parser/
    renderers.

You may add tests, documentation, and dependencies if useful, but keep the package practical
     and easy to run locally.

Do not inspect parent directories or any benchmark/evaluator/contract corpus outside this
    workspace. Do not assume access to any external hidden contract corpus. Do not ask
    questions. Implement, test locally, and leave the final implementation in this
    workspace.
```

Broad refinement prompt.

```
You are working in an existing independent workspace that already contains an improved
    Python Markdown parser/renderer library named `solmarkdown`.

Run a fourth improvement pass. This time, focus on the kinds of last-mile interoperability
    gaps that mature Markdown renderers often differ on.

Keep this public API stable:

- `from solmarkdown import render, render_inline`
- `render(markdown: str | None, *, options: dict | None = None) -> str`
- `render_inline(markdown: str | None, *, options: dict | None = None) -> str`

Hints, in natural language only:

- Pay close attention to CommonMark inline parser edge cases where link or image label
    scanning fails but code spans inside the leftover literal text must still be recognized
    .
- Review inline link destinations around escaped backslashes, spaces, optional titles, and
    URL normalization/escaping.
- Be conservative with math parsing. Dollar-delimited math should not greedily reinterpret
    ordinary text, blank-line-separated paragraphs, or pathological brace-heavy content.
- Wiki-link parsing should avoid converting malformed or nested-Markdown-looking wiki labels
     into links.
- Heading-attribute parsing should be strict about escaped braces, malformed attribute tails
    , and attributes split across lines.

Preserve existing behavior and tests. Add focused public-behavior tests as useful.

Do not assume access to any external hidden contract corpus. Do not ask questions. Implement
    , test locally, and leave the improved implementation in this workspace.
```

B.6 YAML PARSING AND EMISSION

Initial prompt.

```
You are working in an empty independent workspace.

Implement a production-quality, dependency-light Python YAML parser/emitter library named `solyaml`.

Create whatever files are appropriate, but expose this public API:

- `from solyaml import parse, parse_all, dump, dump_all`
- `parse(yaml_text: str | bytes | None, *, options: dict | None = None) -> object`
- `parse_all(yaml_text: str | bytes | None, *, options: dict | None = None) -> list[object]`
- `dump(value: object, *, options: dict | None = None) -> str`
- `dump_all(values: list[object], *, options: dict | None = None) -> str`

Expected behavior:

- Parse YAML streams into deterministic Python values using ordinary data types: `dict`, `list`, `str`, `int`, `float`, `bool`, and `None`.
- Support mature YAML behavior across block and flow mappings/sequences, plain/single/double quoted scalars, literal and folded block scalars, anchors and aliases, merge keys, explicit tags for common YAML scalar types, multiple documents, directives, comments, Unicode input, UTF BOMs, and practical JSON-subset parsing.
- Detect malformed YAML with clear exceptions.
- Emit deterministic YAML for ordinary Python values and preserve semantic round trips through `parse` and `parse_all`.
- Handle common YAML 1.1 and YAML 1.2 scalar resolution behavior pragmatically, especially booleans, nulls, integers, floats, infinities, NaN values, and quoted strings.
- `options` may contain booleans or nested dictionaries for loader/emitter preferences. Treat unknown options safely.
- Serialization should be stable and conventional enough for mature YAML parser/emitters.

You may add tests, documentation, and dependencies if useful, but keep the package practical and easy to run locally.

Do not assume access to any external hidden contract corpus. Do not ask questions. Implement, test locally, and leave the final implementation in this workspace.
```

Broad refinement prompt.

```
You are continuing work in an independent workspace containing a Python YAML parser/emitter library named `solyaml`.

This is a fourth implementation pass. Improve the library while preserving the public API:

- `from solyaml import parse, parse_all, dump, dump_all`
- `parse(yaml_text, *, options=None)`
- `parse_all(yaml_text, *, options=None)`
- `dump(value, *, options=None)`
- `dump_all(values, *, options=None)`

Keep ordinary Python return types and deterministic YAML emission.

Natural-language compatibility hints:

- Pay careful attention to historical YAML fixture styles where a single top-level document is a sequence of mappings, or where mapping entries use unusual-looking keys and nested sequences. These should remain one YAML document when the stream has no document separators.
- Be cautious about cases where wrapper logic around `parse()` might mistake a top-level collection for multiple documents.
- Keep scalar resolution, stream parsing, merge keys, anchors, aliases, duplicate keys, byte/unicode input, malformed-input errors, and round trips compatible with mature YAML libraries.

Do not assume access to any external hidden contract corpus. Do not ask questions. Implement, test locally, and leave the improved implementation in this workspace.
```

B.7 JWT SIGNING AND VERIFICATION

Initial prompt.

```
You are working in an empty independent workspace.

Implement a production-quality, dependency-light Python JWT signing and verification library
     named `soljwt`.

Create whatever files are appropriate, but expose this public API:

- `from soljwt import encode, decode, verify`
- `encode(payload: dict, key: str | bytes | None = None, *, algorithm: str = "HS256",
    headers: dict | None = None, options: dict | None = None) -> str`
- `decode(token: str, key: str | bytes | None = None, *, algorithms: list[str] | None = None
    , verify: bool = True, options: dict | None = None, audience: str | list[str] | None =
    None, issuer: str | list[str] | None = None, subject: str | None = None, leeway: int |
    float = 0, current_time: int | float | None = None, complete: bool = False) -> dict`
- `verify(token: str, key: str | bytes | None = None, **kwargs) -> dict`

Expected behavior:

- Encode and decode compact JWT/JWS strings deterministically.
- Support practical HMAC algorithms HS256, HS384, and HS512.
- Support unsigned `none` tokens only when explicitly requested and never as a permissive
    verification default.
- Preserve and expose protected headers and JSON payload claims when `complete=True`.
- Verify signatures, algorithm allow-lists, and common registered claims: `exp`, `nbf`, `iat
    `, `iss`, `sub`, `aud`, and `jti`.
- Support ordinary custom JSON claims including strings, numbers, booleans, nulls, arrays,
    and nested objects.
- Support useful verification options such as disabling expiration or not-before checks,
    explicit current time, clock tolerance/leeway, required claims, and complete-result
    verification.
- Detect malformed compact tokens, invalid base64url, invalid JSON, unsupported algorithms,
    bad signatures, and invalid claim types with clear exceptions.
- Keep serialization stable and conventional enough for mature JWT libraries.

You may add tests, documentation, and small dependencies if useful, but keep the package
    practical and easy to run locally.

Do not assume access to any external hidden contract corpus. Do not ask questions. Implement
    , test locally, and leave the final implementation in this workspace.
```

Broad refinement prompt.

```
You are continuing work on the existing `soljwt` Python JWT library in this workspace.

Make a targeted production-quality improvement pass. Keep the public API stable:

- `from soljwt import encode, decode, verify`
- `encode(payload: dict, key: str | bytes | None = None, *, algorithm: str = "HS256",
    headers: dict | None = None, options: dict | None = None) -> str`
- `decode(token: str, key: str | bytes | None = None, *, algorithms: list[str] | None = None
    , verify: bool = True, options: dict | None = None, audience: str | list[str] | None =
    None, issuer: str | list[str] | None = None, subject: str | None = None, leeway: int |
    float = 0, current_time: int | float | None = None, complete: bool = False) -> dict`
- `verify(token: str, key: str | bytes | None = None, **kwargs) -> dict`

Natural-language compatibility focus:

- Treat unverified decoding as an observation API for compact JWTs. When verification is
    disabled, it should be able to expose protected headers and JSON payload claims even
    for algorithms this library cannot verify or sign.
- Do not require the signature segment to be cryptographically meaningful, supported by a
    local algorithm implementation, or strictly canonical when verification is disabled.
    Verification mode should remain strict.
- Keep malformed-token rejection for genuinely unreadable headers or payloads.
- Preserve useful complete decode output, including header, payload, and a conventional
    representation of the signature segment.

Do not inspect parent directories or any hidden/external contract corpus. Do not ask
    questions. Work only in this implementation workspace, run your own local tests, and
    leave the improved implementation in place.
```

B.8 CRON AND SCHEDULE EXPRESSIONS

Initial prompt.

```
You are working in an empty independent workspace.

Implement a production-quality, dependency-light Python cron / schedule expression engine
    named `solcron`.

Create whatever files are appropriate, but expose this public API:

- `from solcron import parse, is_valid, normalize, next_dates, prev_dates, match`
- `parse(expression: str, *, options: dict | None = None) -> object`
- `is_valid(expression: str, *, options: dict | None = None) -> bool`
- `normalize(expression: str, *, include_seconds: bool = False, options: dict | None = None)
    -> str`
- `next_dates(expression: str, start: str | object, count: int = 1, *, options: dict | None
    = None) -> list[str]`
- `prev_dates(expression: str, start: str | object, count: int = 1, *, options: dict | None
    = None) -> list[str]`
- `match(expression: str, date: str | object, *, options: dict | None = None) -> bool`

Expected behavior:

- Parse ordinary 5-field crontab expressions and practical 6-field expressions with seconds.
- Support wildcards, lists, ranges, steps, month/day names, whitespace normalization, and
    common validation errors.
- Compute deterministic next and previous run dates from ISO-8601 start times.
- Return date strings in stable ISO-8601 UTC form when possible.
- Support matching a date against an expression.
- Handle practical timezone-aware ISO inputs and naive ISO inputs consistently.
- Treat unknown options safely; useful options may include timezone, day-of-month/day-of-
    week matching policy, and second-field position.
- Detect malformed expressions with clear exceptions.
- Keep the library simple to install and easy to run locally.

You may add tests, documentation, and small dependencies if useful, but keep the package
    practical.

Do not assume access to any external hidden contract corpus. Do not ask questions. Implement
    , test locally, and leave the final implementation in this workspace.
```

Broad refinement prompt.

```
You are working in an independent workspace containing the current `solcron` Python package.

Continue improving the implementation as a mature, production-quality cron / schedule
    expression engine while preserving the public API:

- `from solcron import parse, is_valid, normalize, next_dates, prev_dates, match`
- `parse(expression: str, *, options: dict | None = None) -> object`
- `is_valid(expression: str, *, options: dict | None = None) -> bool`
- `normalize(expression: str, *, include_seconds: bool = False, options: dict | None = None)
    -> str`
- `next_dates(expression: str, start: str | object, count: int = 1, *, options: dict | None
    = None) -> list[str]`
- `prev_dates(expression: str, start: str | object, count: int = 1, *, options: dict | None
    = None) -> list[str]`
- `match(expression: str, date: str | object, *, options: dict | None = None) -> bool`

Focus especially on compatibility details that mature cron libraries tend to support or
    intentionally validate:

- Quartz/Spring-style expressions may include an optional year field, use `?` in day fields,
    and combine named months/weekdays with the year field.
- Matching should behave consistently at cron field precision even when the input datetime
    carries fractional seconds.
- Nearest-weekday `W` handling should be strict about forms that are conventionally
    unsupported or ambiguous, especially when `W` appears inside comma-separated day-of-
    month lists or mixed with ordinary days.

Do not assume access to any external hidden contract corpus. Do not ask questions. Implement
    , test locally, and leave the improved implementation in this workspace.
```

B.9 RETRY, BACKOFF, AND RESILIENCE POLICIES

Initial prompt.

```
You are working in an empty independent workspace.

Implement a production-quality, dependency-light Python retry / backoff / resilience policy
     library named `solretry`.

Create whatever files are appropriate, but expose this public API:

- `from solretry import retry, RetryPolicy, retryable`
- `retry(fn, *, max_attempts: int = 3, wait: float = 0, retry_exceptions: tuple[type[
     BaseException], ...] | type[BaseException] = Exception, retry_on_result=None, on_retry=
     None, raise_on_giveup: bool = True, **kwargs) -> object`
- `RetryPolicy(max_attempts: int = 3, wait: float = 0, retry_exceptions=Exception,
     retry_on_result=None, on_retry=None, raise_on_giveup: bool = True, **kwargs)`
- `RetryPolicy.call(fn, *args, **kwargs) -> object`
- `retryable(...)` as a decorator factory using the same options as `retry`.

Expected behavior:

- Retry callables that raise configured exceptions until they succeed or the attempt limit
     is reached.
- Treat `max_attempts` as total attempts, including the first attempt.
- Validate attempt limits sensibly; values below one should behave as one attempt.
- Do not retry exceptions outside the configured retry exception set.
- Preserve and re-raise the final exception on give-up by default.
- Support zero wait and small fixed waits without requiring real long sleeps.
- Call `on_retry` after failed attempts that will be retried, passing useful information
     such as the exception and attempt number.
- Return successful values unchanged, including falsey values.
- Provide a small, practical API surface rather than a large framework.
- Keep behavior deterministic and easy to test locally.
- Add tests or examples if helpful.

Do not assume access to any external hidden contract corpus. Do not ask questions. Implement
     , test locally, and leave the final implementation in this workspace.
```

Broad refinement prompt.

```
Continue improving the `solretry` implementation in this workspace.

This is iteration 6, starting from the iteration 5 implementation. Improve the
     implementation using your own engineering judgment. You may use the following broad
     failure-area hints, but you must not assume access to any hidden benchmark or contract
     corpus.

Natural-language hints only:

- Retry semantics should treat the configured maximum as total call attempts, including the
     first call. A handled exception should not stop the operation immediately; retry until
     success or until the configured attempt boundary is reached.
- Make exception classification central and consistent. Only configured/handled exception
     classes should be retried; unhandled exception classes should propagate immediately.
- Retry exhaustion should preserve the final failure shape, while successful later attempts
     should return the successful value.
- Retry callbacks/listeners should be invoked only for scheduled retry attempts, not for the
      final exhausted failure. They should expose a useful attempt number and exception/
     result context.
- Support the common API vocabulary used across mature retry libraries: `max_attempts`, `
     max_retries`, `tries`, `max_tries`, fixed waits, zero waits, retry-on-result/predicate
     behavior, and decorator/wrapper surfaces.
- Avoid real sleeping in tests and allow the sleep/wait mechanism to be injected or observed
     .

Do not read files outside this implementation workspace. Do not look for benchmark files,
     contract files, or previous evaluation output. Run your own tests before finishing.
```

Table 5: Model–framework configurations used in the primary evaluation. The `openai/` prefix used by OpenHands for MiniMax and GLM denotes the LiteLLM API route rather than the model provider.

Framework	Model	Model Identifier	Run Dates
Codex CLI	GPT-5.6-Sol	`gpt-5.6-sol`	2026-08-29–31
	MiniMax M3	`MiniMax-M3`	2026-09-02–04
	GLM 5.3	`glm-5.3`	2026-09-04, 06–07
Claude Code	Claude Opus 5	`claude-opus-5`	2026-09-04–06
	MiniMax M3	`MiniMax-M3`	2026-09-08
	GLM 5.3	`glm-5.3`	2026-09-07
OpenHands	GPT-5.6-Sol	`openai/gpt-5.6-sol`	2026-09-08–09
	Claude Opus 5	`anthropic/claude-opus-5`	2026-09-09–10
	MiniMax M3	`openai/MiniMax-M3`	2026-09-11
	GLM 5.3	`openai/glm-5.3`	2026-09-12–13

Table 6: Recorded agent-framework versions and primary execution settings.

Framework	Version	Primary Execution Configuration
Codex CLI	0.1.2505172129	Non-interactive `exec`; reasoning effort `high`; approval disabled; user configuration ignored; Git-repository check skipped; typically full-access sandbox.
Claude Code	2.1.260	Non-interactive `claude -p`; effort `high`; `bypassPermissions`; streamed JSON output; project/local settings enabled.
OpenHands	CLI 1.16.0 / SDK 1.21.0	Headless JSON execution with automatic approval and environment overrides; provider and timeout settings supplied through the child-process environment.

C Models, Agent Frameworks, and Execution Configuration

The primary evaluation contains 10 supported model–framework configurations rather than a complete 4-by-3 factorial. Table 5 reports the provider/model identifier and the dates observed in the preserved primary run records. The dates identify experiment executions rather than immutable provider model snapshots.

Each session operated on the source files produced by previous sessions but started from a fresh interaction context. Agents could edit files, execute shell commands, install dependencies, build and test their implementations, and use the native toolchain exposed by the framework. Network access through the execution environment was not globally disabled. Hidden behavioral evaluation was executed outside the active agent interaction. Provider credentials and model-selection variables were injected only into the child process used for each run. Reasoning effort was set to `high` throughout the primary evaluation.

D Per-Domain Agentic Development Trajectories

Figure 3 in the main paper aggregates development trajectories by framework and model. We additionally decompose these averages by software domain. The underlying analysis contains 90 model–framework–domain trajectories. For each trajectory, the first four observed checkpoints and the final checkpoint are retained, yielding 450 selected checkpoint records. Cumulative agentic steps are summed within each trajectory before aggregation. The resulting trajectories are observational and are not normalized to equal compute.

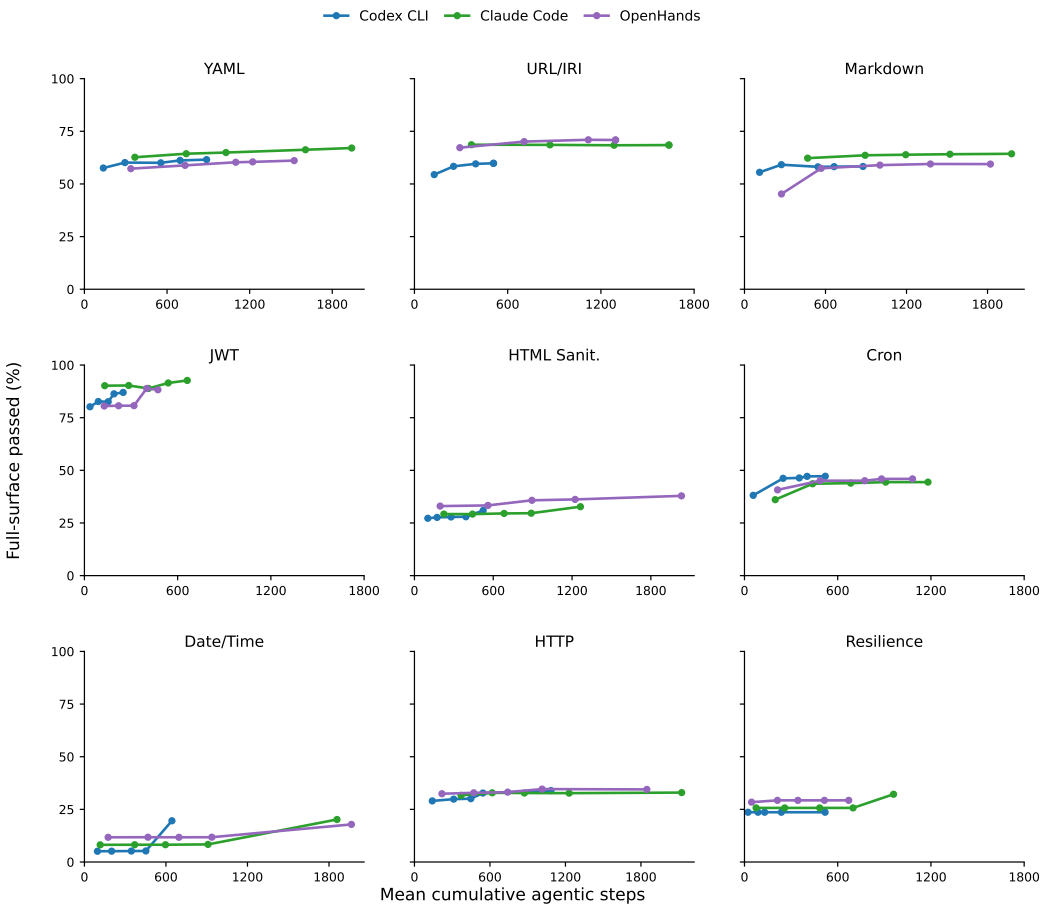


Figure 5: Full-surface behavioral coverage over cumulative agentic steps, decomposed by domain and agent framework. Each panel reports framework-level means within one domain using the same selected checkpoints as Figure 3.

The domain-level trajectories show that the aggregate saturation pattern is not driven by a single software domain. JWT reaches high final coverage across frameworks and models, whereas Date/Time remains substantially lower despite longer trajectories. Relative framework and model ordering also changes across domains, reinforcing that larger agentic-step counts do not correspond monotonically to greater behavioral coverage.

E Recurrence-Specific Failure Patterns by Model and Domain

Figure 4 in the main paper aggregates residual failures across all evaluated configurations. We further decompose the same analysis by model and by software domain using the identical recurrence strata and relative miss-enrichment definition. The larger row-wise layouts below preserve the labels and numerical annotations that become difficult to read in a single composite figure.

Figures 7 and 8 show the model-level breakdown. The overall recurrence gradient remains visible across all four models, although the strength of miss enrichment varies by domain.

Figures 9–11 provide the complementary domain-level view, comparing models within each software domain. The aggregate pattern from the main paper persists across finer slices, but the magnitude of enrichment and the residual high-recurrence failures differ substantially across domains.

Across the model-specific views, the broad recurrence gradient remains visible despite differences in magnitude. The domain-specific views show a similar pattern while also revealing substantial

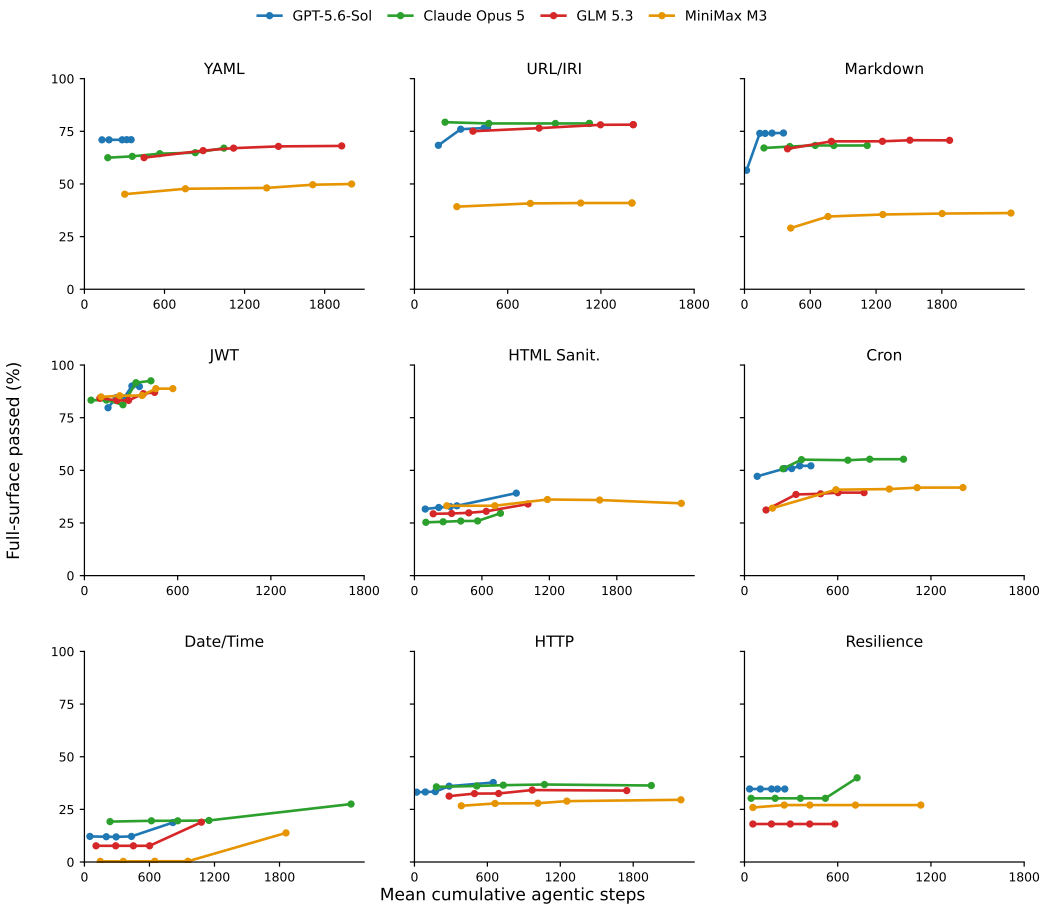


Figure 6: Full-surface behavioral coverage over cumulative agentic steps, decomposed by domain and model. Model means aggregate only the framework pairings actually evaluated for that model.

variation across software domains and models. Together, these decompositions support the interpretation of recurrence as a broad signal of realizability rather than an effect driven by a single model or domain.

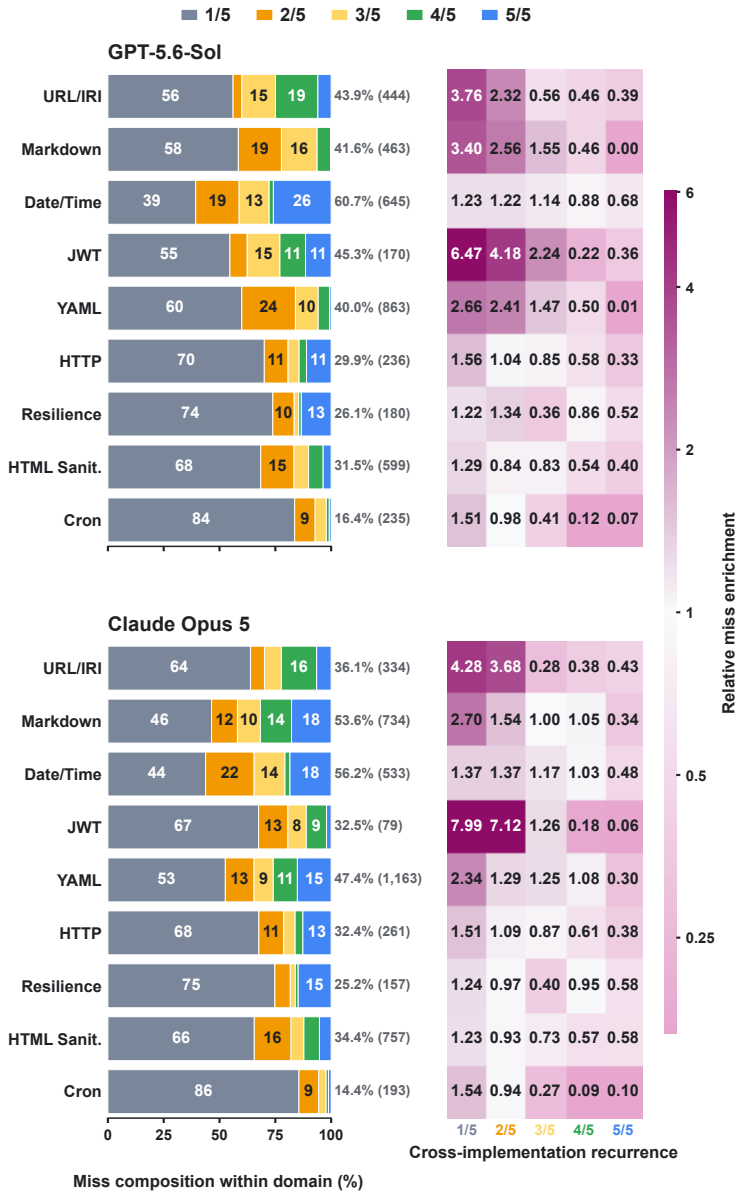


Figure 7: Agent misses by cross-implementation recurrence for GPT-5.6-Sol and Claude Opus 5.

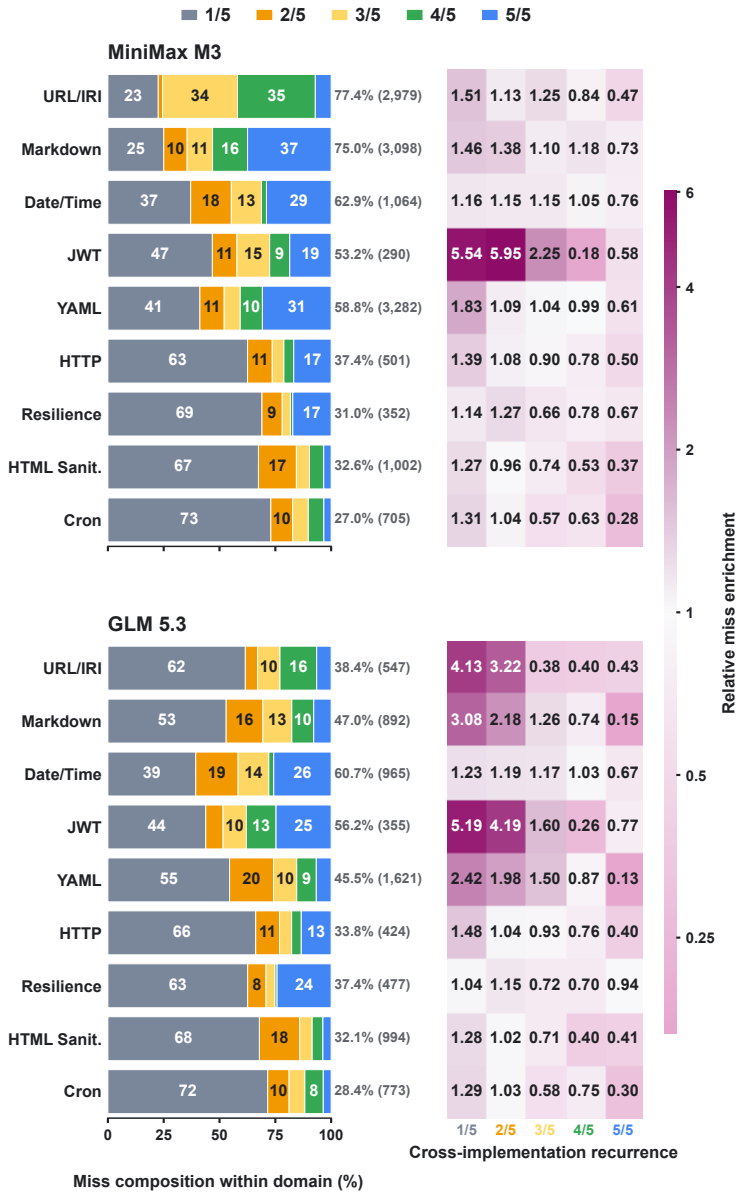


Figure 8: Agent misses by cross-implementation recurrence for MiniMax M3 and GLM 5.3.

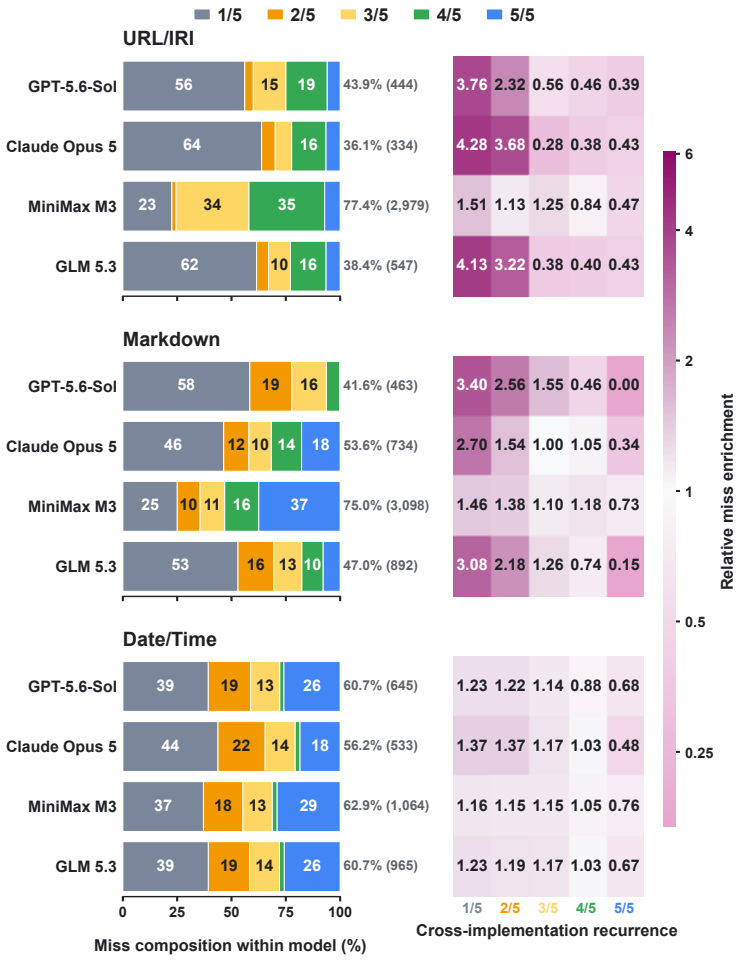


Figure 9: Model-specific failure patterns for URL/IRI, Markdown, and Date/Time.

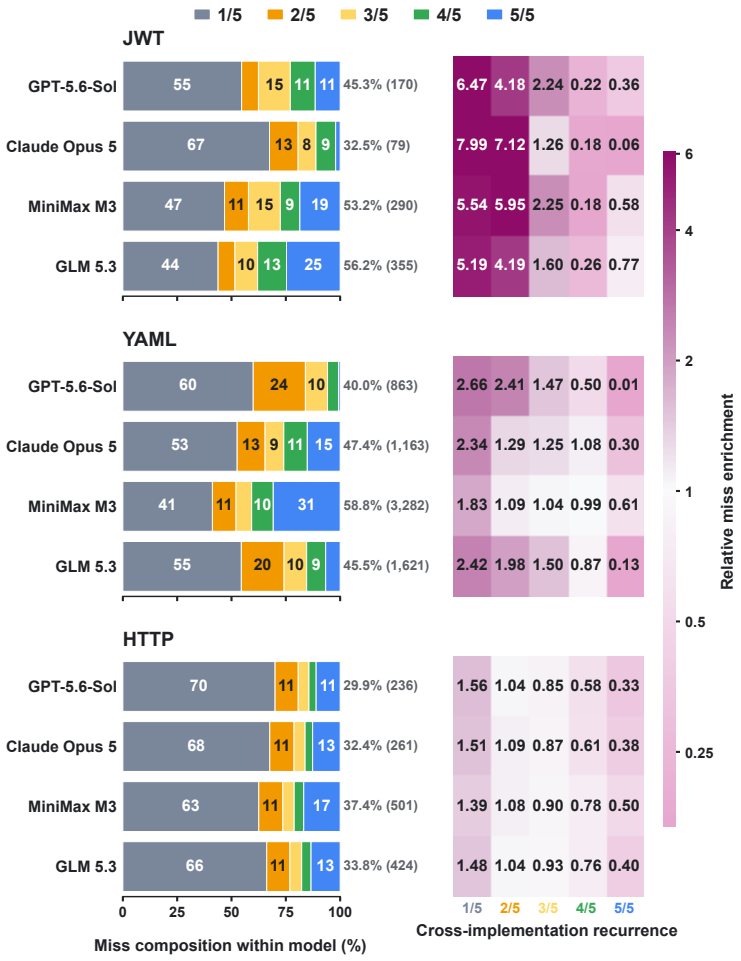


Figure 10: Model-specific failure patterns for JWT, YAML, and HTTP.

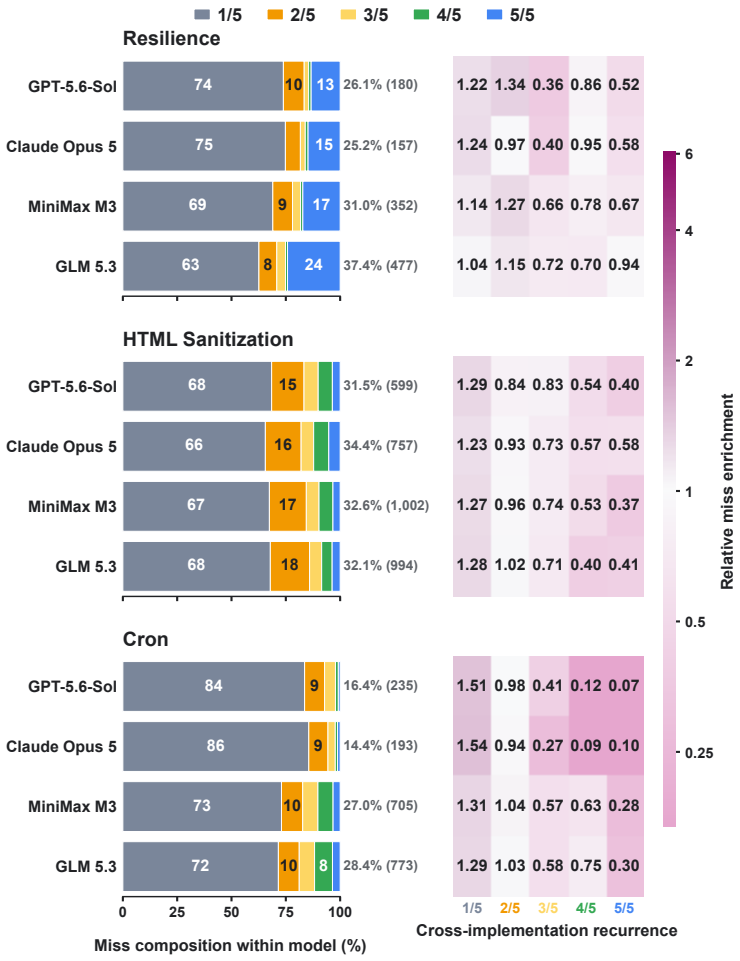


Figure 11: Model-specific failure patterns for Resilience, HTML Sanitization, and Cron.

F End-to-End Contract Construction and Validation Example

We illustrate the executable-contract pipeline with a URL/IRI behavior recovered from `jsdom/whatwg-url`. The selected contract is `1.0.0:url_constructor_resolves_relative_against_base`, with canonical scoring identifier `url_iri:core:whatwg-url:1.0.0:url_constructor_resolves_relative_against_base`. The behavior requires the relative input `../d?q=1` against base `https://example.com/a/b/c` to resolve to `https://example.com/a/d?q=1`.

F.1 Evidence Recovery and Contract Construction

The contract is created directly from recovered release evidence. The following is the actual extraction code used for this behavior.

```
def pass2(version: str, body: str) -> list[Contract]:
    b = body.lower()
    contracts: list[Contract] = []

    if "url constructor" in b or "overhauled api" in b:
        c(contracts, version, body, "url_constructor_exposes_components", "url.api.
     constructor-components", "parse_url",
          {"input": "https://user:pass@example.com:8443/a/b?x=1#frag"}, {"protocol": "https
     :", "username": "user", "password": "pass", "hostname": "example.com", "port": "8443",
     "pathname": "/a/b", "search": "?x=1", "hash": "#frag"}, "
     constructor_does_not_expose_components", "pass2")
        c(contracts, version, body, "url_constructor_resolves_relative_against_base", "url.
     api.constructor-base", "parse_url",
          {"input": "../d?q=1", "base": "https://example.com/a/b/c"}, {"href": "https://
     example.com/a/d?q=1"}, "constructor_ignores_base_url", "pass2")
```

The resulting source artifact preserves the contract fields and release evidence:

```
      "name": "1.0.0:url_constructor_resolves_relative_against_base",
      "version": "1.0.0",
      "capability": "url.api.constructor-base",
      "op": "parse_url",
      "params": {
        "input": "../d?q=1",
        "base": "https://example.com/a/b/c"
      },
      "expected": {
        "href": "https://example.com/a/d?q=1"
      },
      "mutant": "constructor_ignores_base_url",
      "evidence": "GitHub release v1.0.0: Overhauled API to expose the `URL` constructor
     alongside the lower-level algorithms from the URL Standard.",
      "pass": "pass2"
    },
```

The executable RPL representation is:

```
contract 1.0.0:url_constructor_resolves_relative_against_base {
  version = "1.0.0"
  capability = "url.api.constructor-base"
  op = "parse_url"
  params = {"base": "https://example.com/a/b/c", "input": "../d?q=1"}
  expect = {"href": "https://example.com/a/d?q=1"}
  mutant = "constructor_ignores_base_url"
}
```

F.2 Normalization into the Domain-Level Representation

Before cross-implementation replay, expected observations are projected to the common URL/IRI representation. The actual normalization logic is:

```
def pick_keys(data: dict[str, Any], keys: list[str] = COMMON_KEYS) -> dict[str, Any]:
    return {key: data[key] for key in keys if key in data}


def normalize_expected_from_parse(expected: dict[str, Any]) -> dict[str, Any]:
    out = pick_keys(expected)
    if "ok" not in out:
        out["ok"] = True
    return out
```

```
def portable(origin: str, contract: dict[str, Any]) -> dict[str, Any] | None:
    op = contract["op"]
    params = copy.deepcopy(contract.get("params") or {})
    expected = contract.get("expected") or {}

    if origin in {"whatwg-url", "rust-url"}:
        if op == "parse_url":
            return {"kind": "parse", "params": params, "expected": compact_expected(
     normalize_expected_from_parse(expected))}
```

The preserved normalized RPL contract is:

```
contract whatwg-url::1.0.0:url_constructor_resolves_relative_against_base {
  origin = "whatwg-url"
  version = "1.0.0"
  capability = "url.api.constructor-base"
  kind = "parse"
  params = {"base": "https://example.com/a/b/c", "input": "../d?q=1"}
  expect = {"href": "https://example.com/a/d?q=1", "ok": true}
}
```

F.3 Source Adapter and Positive Validation

For `whatwg-url`, the adapter constructs the native `URL` object and projects its externally observable properties into the common representation:

```
const w = require("./node_modules/whatwg-url");

function urlFrom(input, base) {
  return base === undefined ? new w.URL(input) : new w.URL(input, base);
}

function publicURL(u) {
  return {
    href: u.href,
    origin: u.origin,
    protocol: u.protocol,
    username: u.username,
    password: u.password,
    host: u.host,
    hostname: u.hostname,
    port: u.port,
    pathname: u.pathname,
    search: u.search,
    hash: u.hash
  };
}
```

```
function execute(contract) {
  const p = contract.params || {};
  switch (contract.op) {
    case "parse_url":
      return publicURL(urlFrom(p.input, p.base));
```

```
def run_node(contract: dict[str, Any]) -> dict[str, Any]:
    proc = subprocess.run(
        ["node", "-e", NODE_RUNNER],
        cwd=PACKAGE_DIR,
        input=json.dumps(contract),
        text=True,
        capture_output=True,
        check=False,
        timeout=10,
    )
    if proc.returncode != 0:
        return {"ok": False, "error": {"name": "NodeProcessError", "message": proc.stderr.
     strip()}}
    try:
        return json.loads(proc.stdout)
    except json.JSONDecodeError as exc:
        return {"ok": False, "error": {"name": "JSONDecodeError", "message": str(exc), "
     stdout": proc.stdout, "stderr": proc.stderr}}
```

The oracle compares only the fields specified by the contract:

```
def expected_matches(actual: dict[str, Any], expected: dict[str, Any]) -> tuple[bool, list[
     str]]:
    misses: list[str] = []
    for key, value in expected.items():
        if key.endswith("Contains"):
            actual_key = key[: -len("Contains")]
            actual_value = actual.get(actual_key)
            if isinstance(actual_value, list):
                if value not in actual_value:
                    misses.append(f"{key}: {value!r} not in {actual_value!r}")
            elif isinstance(actual_value, str):
                if value not in actual_value:
                    misses.append(f"{key}: {value!r} not in {actual_value!r}")
            else:
                misses.append(f"{key}: actual {actual_key!r} is not searchable ({
     actual_value!r})")
        elif actual.get(key) != value:
            misses.append(f"{key}: expected {value!r}, got {actual.get(key)!r}")
    return not misses, misses
```

The validation loop executes the contract, evaluates the source replay, perturbs the expected observation, and retains the contract only when both the replay and oracle sanity checks succeed:

```
    survivors = []

    for contract in contracts:
        node_result = run_node(contract)
        if not node_result.get("ok"):
            replay_ok = False
            actual = {}
            failures = [node_result.get("error", {}).get("message", "unknown node failure")]
        else:
            actual = node_result["actual"]
            replay_ok, failures = expected_matches(actual, contract["expected"])

        mutant_expected = mutate_expected(contract["expected"])
        mutant_ok, mutant_failures = expected_matches(actual, mutant_expected)
        mutant_killed = not mutant_ok
        survived = replay_ok and mutant_killed
```

F.4 EXPECTATION-PERTURBATION ORACLE SANITY CHECK

The oracle sanity check perturbs the expected observation while leaving the reference implementation and observed output unchanged. The purpose is to verify that the oracle is sensitive to changes in the scored expectation, rather than to establish corpus-wide mutation adequacy. The implementation is:

```
def mutate_value(value: Any) -> Any:
    if isinstance(value, bool):
        return not value
    if isinstance(value, int):
        return value + 1
    if isinstance(value, str):
        return value + "__mutant__"
    if isinstance(value, list):
        return list(reversed(value)) if len(value) > 1 else value + ["__mutant__"]
    if value is None:
        return "__mutant__"
    if isinstance(value, dict):
        mutated = copy.deepcopy(value)
        if mutated:
            first = next(iter(mutated))
            mutated[first] = mutate_value(mutated[first])
        else:
            mutated["__mutant__"] = True
        return mutated
    return "__mutant__"


def mutate_expected(expected: dict[str, Any]) -> dict[str, Any]:
    mutated = copy.deepcopy(expected)
    key = next(iter(mutated))
    mutated[key] = mutate_value(mutated[key])
    return mutated
```

For the worked contract, the preserved replay result records both the successful positive observation and rejection of the mutated expectation:


```
      "name": "1.0.0:url_constructor_resolves_relative_against_base",
      "version": "1.0.0",
      "capability": "url.api.constructor-base",
      "op": "parse_url",
      "replay_ok": true,
      "mutant_killed": true,
      "survived_latest": true,
      "expected": {
        "href": "https://example.com/a/d?q=1"
      },
      "actual": {
        "href": "https://example.com/a/d?q=1",
        "origin": "https://example.com",
        "protocol": "https:",
        "username": "",
        "password": "",
        "host": "example.com",
        "hostname": "example.com",
        "port": "",
        "pathname": "/a/d",
        "search": "?q=1",
        "hash": ""
      },
      "failures": [],
      "mutant_expected": {
        "href": "https://example.com/a/d?q=1__mutant__"
      },
      "mutant_failures": [
        "href: expected 'https://example.com/a/d?q=1__mutant__', got 'https://example.com/a/
    d?q=1'"
      ]
```

F.5 CROSS-IMPLEMENTATION REPLAY

After source validation, the same normalized contract is dispatched to each reference implementation through implementation-specific adapters. Target dispatch is explicit:

```
def run_target(target: str, items: list[dict[str, Any]]) -> list[dict[str, Any]]:
    if target == "whatwg-url":
        return run_node_batch(items)
    if target == "yarl":
        return run_yarl_batch(items)
    if target == "rust-url":
        return [run_rust_one(item) for item in items]
    if target == "ada-url":
        return [run_ada_one(item) for item in items]
    if target == "curl":
        return [run_curl_one(item) for item in items]
    raise ValueError(target)
```

The common oracle and aggregation loop are:

```
def expected_matches(expected: dict[str, Any], actual: dict[str, Any]) -> tuple[bool, list[
    str]]:
    misses = []
    if actual.get("unsupported"):
        return False, [f"unsupported: {actual.get('message')}"]
    for key, value in expected.items():
        if actual.get(key) != value:
            misses.append(f"{key}: expected {value!r}, got {actual.get(key)!r}")
    return not misses, misses
```

```
def build_origin(origin: str, contracts: list[dict[str, Any]], targets: list[str]) -> dict[
    str, Any]:
    candidates = []
    unsupported = []
    for contract in contracts:
        p = portable(origin, contract)
        if p is None or not p.get("expected"):
            unsupported.append(contract["name"])
            continue
        p = {**p, "origin": origin, "contract": contract}
        candidates.append(p)

    target_results: dict[str, list[dict[str, Any]]] = {}
    for target in targets:
        target_results[target] = run_target(target, candidates)

    survivors = []
    failures = []
    for index, candidate in enumerate(candidates):
        per_target = {}
        passed = True
        for target in targets:
            actual = target_results[target][index]
            ok, misses = expected_matches(candidate["expected"], actual)
            per_target[target] = {"passed": ok, "actual": actual, "misses": misses}
            passed = passed and ok
        row = {
            "origin": origin,
            "name": candidate["contract"]["name"],
            "version": candidate["contract"]["version"],
```

The first three implementations are used to form portable candidates, after which the remaining two implementations are applied as additional cross-implementation filters:

```
def main() -> None:
    OUT_DIR.mkdir(parents=True, exist_ok=True)
    origins = {
        "whatwg-url": load(WHATWG_JSON),
        "rust-url": load(RUST_JSON),
        "yarl": load(YARL_JSON),
    }
    stage_123 = [
        build_origin("whatwg-url", origins["whatwg-url"], ["rust-url", "yarl"]),
        build_origin("rust-url", origins["rust-url"], ["whatwg-url", "yarl"]),
```

```
        build_origin("yarl", origins["yarl"], ["whatwg-url", "rust-url"]),
    ]
    raw_common_candidates = [row for stage in stage_123 for row in stage["survivors"]]
    common_candidates = dedupe_rows(raw_common_candidates)
    filter4 = run_target("ada-url", common_candidates)
    filter5 = run_target("curl", common_candidates)
    final_common = []
    hidden_failures = []
    for row, ada_actual, curl_actual in zip(common_candidates, filter4, filter5, strict=True
    ):
        ada_ok, ada_misses = expected_matches(row["expected"], ada_actual)
        curl_ok, curl_misses = expected_matches(row["expected"], curl_actual)
        enriched = {
            **row,
            "hidden_filters": {
                "ada-url": {"passed": ada_ok, "actual": ada_actual, "misses": ada_misses},
                "curl": {"passed": curl_ok, "actual": curl_actual, "misses": curl_misses},
            },
        }
        if ada_ok and curl_ok:
            final_common.append(enriched)
        else:
            hidden_failures.append(enriched)
```

Representative native adapters. The same `kind=parse` behavior is bound to each implementation's native URL API. The following excerpts show the actual bindings used for the worked case.

whatwg-url:

```
const w = require("./node_modules/whatwg-url");

function urlFrom(input, base) {
  return base === null || base === undefined ? new w.URL(input) : new w.URL(input, base);
}

function common(u) {
  return {
    ok: true,
    href: u.href,
    origin: u.origin,
    protocol: u.protocol,
    username: u.username,
    password: u.password,
    host: u.host,
    hostname: u.hostname,
    port: u.port,
    pathname: u.pathname,
    search: u.search,
    hash: u.hash
  };
}

function runOne(item) {
  const p = item.params || {};
  try {
    if (item.kind === "parse") {
      return common(urlFrom(p.input, p.base));
    }
```

yarl:

```
YARL_BATCH = r"""
import json
import sys
from yarl import URL

def common(u):
    raw_host = u.raw_host or ""
    explicit_port = u.explicit_port
    port = "" if explicit_port is None else str(explicit_port)
    host = raw_host
    if port:
        host = f"{host}:{port}" if host else f":{port}"
    query_string = u.raw_query_string
    fragment = u.raw_fragment
    return {
        "ok": True,
        "href": str(u),
        "origin": _origin(u),
        "protocol": f"{u.scheme}:" if u.scheme else "",
        "username": u.raw_user or "",
        "password": u.raw_password or "",
        "host": host,
        "hostname": raw_host,
        "port": port,
        "pathname": u.raw_path,
        "search": f"?{query_string}" if query_string else "",
        "hash": f"#{fragment}" if fragment else "",
    }

def _origin(u):
    try:
        return str(u.origin())
    except Exception:
        return "null"

def make_url(p):
    if p.get("base"):
        return URL(p["base"]).join(URL(p["input"]))
    return URL(p["input"])
```

```
def run_one(item):
    p = item.get("params") or {}
    try:
        if item["kind"] == "parse":
            return common(make_url(p))
```

rust-url:

```
fn parse_url(params: &Value) -> Result<Value, String> {
    let input = req_str(params, "input")?;
    if let Some(base) = opt_str(params, "base") {
        let base = Url::parse(base).map_err(|e| format!("{e:?}"))?;
        base.join(input)
            .map(|url| url_summary(&url))
            .map_err(|e| format!("{e:?}"))
    } else {
        Url::parse(input)
            .map(|url| url_summary(&url))
            .map_err(|e| format!("{e:?}"))
    }
}
```

ada-url:

```
fn parse_url(params: &Value) -> Value {
    let input = s(params, "input").unwrap_or("");
    let base = s(params, "base");
    match Url::parse(input, base) {
        Ok(url) => json!({"ok": true, "result": url_props(&url)}),
        Err(err) => json!({"ok": false, "throws": "ParseUrlError", "message": err.to_string
     ()}),
    }
```

curl URL API:

```
static void op_relative_hex(int argc, char **argv)
{
  char *base_url = hex_arg(argv[2]);
  char *relative_url = hex_arg(argv[4]);
  CURLU *h = curl_url();
  CURLUcode base = curl_url_set(h, CURLUPART_URL, base_url, uint_arg(argv[3]));
  CURLUcode rel = curl_url_set(h, CURLUPART_URL, relative_url, uint_arg(argv[5]));
  unsigned int get_flags = uint_arg(argv[6]);
  putchar('{');
  code_field("base", base);
  putchar(',');
  code_field("relative", rel);
  if(!base && !rel) {
    putchar(',');
    emit_parts(h, get_flags);
  }
  putchar('}');
  curl_url_cleanup(h);
  free(base_url);
  free(relative_url);
  (void)argc;
}
```

The preserved observations for this contract are identical across all five references, yielding a recurrence count of $s_d(c) = 5$ and placing the contract in the *Shared Core*.

This worked example shows the full path from release evidence to an executable behavioral contract, source validation, oracle sanity checking, implementation-specific execution, and cross-implementation recurrence. The behavioral requirement itself remains fixed; adapters account for differences in native APIs and return observations in the common representation used by the oracle.

Table 7: Cross-implementation replay of the worked URL-resolution contract. All five reference implementations satisfy the same normalized behavioral requirement.

Reference	Version	Observed `href`
whatwg-url	17.1.0	`https://example.com/a/d?q=1`
rust-url	2.5.8	`https://example.com/a/d?q=1`
yarl	1.24.5	`https://example.com/a/d?q=1`
ada-url	4.0.0	`https://example.com/a/d?q=1`
curl	8.21.0	`https://example.com/a/d?q=1`

G Behavioral-Mutant Validity Audit

In addition to the source-replay and oracle-sanity checks used during contract construction, we performed a behavioral-mutant validity audit without modifying any benchmark contract, expected observation, oracle, or score. Across the 14,537 scoring rows, 13,330 had at least one valid mutant. Restricting the analysis to mutants that changed a contract-required observable behavior yields 22,587 valid mutants, of which 21,861 were rejected by the unchanged scoring oracles, for a conditional detection rate of 96.8%.

The remaining 726 mutants survived the scoring oracle and affect 284 distinct scoring rows. These cases arise from several known sources, including intentionally relaxed exception matching, canonicalization that collapses distinct outputs, and coarse scoring projections that omit fields present in the recovered contract. We retain these cases as explicit audit findings rather than removing or repairing them post hoc.

This audit is not a proof of mutation adequacy. The operator catalog is finite, 535 rows remain blocked, and 672 rows have no applicable operator in the preserved catalog. Masked mutants that do not change the scored required behavior are reported separately and excluded from the detection-rate denominator.