ARC-GOLF BENCH: BENCHMARKING CONTINUAL PROGRAM OPTIMIZATION WITH CODE GOLF AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Code benchmarks typically reward correctness, leaving agents' ability to improve already-correct programs underexplored. We introduce ARC-Golf Bench, a compact benchmark of 96 ARC-AGI grid-transformation tasks from the NeurIPS 2025 Google Code Golf Championship. Guided by item response theory, we select 16 easy, 48 medium, and 32 hard tasks to enable efficient evaluation across a broad difficulty range. Agents minimize Python program length under executable verification, providing an exact, inexpensive objective anchored to human performance. We evaluate four models with two agentic harnesses, tracking the shortest verified solutions as output tokens accumulate. Across configurations, agents improve on the best human reference solutions on ten tasks, but still lag behind the strongest human teams overall, especially on hard tasks. Moreover, performance depends on the model–harness pairing: Codex achieves better human-equivalent ranks for Qwen, GPT, and DeepSeek, while GLM performs better under Claude Code. To characterize the behavior behind these outcomes, we introduce *Golf-UnZip*, which measures golf-trick richness and solution-method diversity. We further investigate experience transfer through a library of reusable simplifications, conditional rewrites, and representation strategies. On ten held-out tasks, library-assisted optimization reduces total program length by 13.2% for one worker and 9.4% for another with an adapted harness, demonstrating transfer across tasks and receiving models. ARC-Golf Bench provides a testbed for studying sustained program optimization and how model capabilities, harness design, and transferred experience shape code continual optimization.

1 INTRODUCTION

Widely used benchmarks such as HumanEval and SWE-bench evaluate functional correctness and successful issue resolution, respectively (Chen et al., 2021; Jimenez et al., 2024). These objectives are essential, but pass/fail outcomes do not distinguish among accepted programs of different quality. Program optimization extends the challenge: an agent must preserve required behavior while searching for better implementations, as in superoptimization and performance-improving code edits (Schkufza et al., 2013; Shypula et al., 2024).

We study the program optimization problem through *code golf*: constructing correct programs while minimizing their length in bytes. Our benchmark builds on the NeurIPS 2025 Google Code Golf Championship (Moffitt et al., 2025), whose tasks involve grid transformations from the Abstraction and Reasoning Corpus (ARC; Chollet, 2019). Given task examples and access to a verifier, agents search for compact Python implementations through algorithmic redesign, representation changes, and local rewrites. Code golf offers an exact, inexpensive objective in a lightweight execution environment, yet accepted programs can remain far longer than the best human solutions. This makes it a useful setting for studying *continual program optimization*: repeated solution refinement and the reuse of optimization experience across problems.

We introduce **ARC-Golf Bench**, a compact benchmark designed to support efficient evaluation across a broad difficulty range. We curate 96 tasks from the competition's 400 problems using a graded response model from item response theory (IRT), fitted to human solution quality. Selection prioritizes tasks that discriminate among human teams while retaining challenging problems. We categorize the selected tasks by their estimated difficulty into 16 easy, 48 medium, and 32 hard tasks.

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.

We evaluate four models with Codex and Claude Code, yielding eight model–harness configurations. We track the shortest verified programs and their human-relative ranks as output tokens accumulate. These trajectories distinguish rapid initial progress from sustained compression. Performance depends strongly on the model–harness pairing: Codex achieves better ranks for Qwen, GPT, and DeepSeek, while GLM performs better under Claude Code. Across configurations, agents improve on the best human reference solutions on ten tasks, but remain behind the strongest human teams overall, especially on hard tasks.

To understand the behavior behind these outcomes, we introduce *Golf-UnZip*. It incrementally expands compressed programs through verified rewrites, exposing local golf tricks and making broader solution methods easier to identify. We use the recovered trick families and task-specific method signatures to characterize golf-trick richness and solution-method diversity. We also measure the fraction of final-file updates linked to successful verification records. Together, these measurements provide behavioral profiles that complement performance trajectories, distinguishing the variety of programs agents discover from their verification practices.

Finally, we investigate experience transfer across problems and receiving models. Transfer is challenging because highly compressed solutions often entangle reusable insights with task-specific assumptions. We distill solutions from 50 source tasks into a library of simplification skills. Each skill specifies when it applies and what computation or code structure it can eliminate; the entire library is supplied in context. On ten held-out tasks, library-assisted optimization reduces total program length by 13.2% for one worker. For another worker, combining the library with a harness that assigns parallel sessions to distinct solution families yields a 9.4% reduction relative to a matched library-free parallel control. These results demonstrate the potential for transferring optimization experience while highlighting the interaction of the receiving model and harness in making that experience useful. Our contributions are as follows:

- We curate ARC-Golf Bench, a compact 96-task benchmark for continual program optimization. IRT-guided selection prioritizes discriminative tasks and spans 16 easy, 48 medium, and 32 hard problems.
- We evaluate four models with two agentic harnesses, revealing evident model–harness interactions. Agents remain behind the strongest human competitors on aggregate but improve on the best human reference solutions on 10 tasks.
- We introduce Golf-UnZip to analyze golf solutions and characterize golf-trick richness and solution-method diversity, providing behavioral profiles of model–harness configurations.
- We investigate optimization-experience transfer across tasks and receiving models, showing that structured skills, together with harness adaptation where needed, reduce total program length by 13.2% and 9.4% for two workers on held-out tasks.

2 Related Work

Code benchmarks span function synthesis, competitive programming, and repository-level software engineering. HumanEval evaluates functional correctness (Chen et al., 2021), EvalPlus strengthens test coverage (Liu et al., 2023), LiveCodeBench uses temporally updated contest problems (Jain et al., 2025), and SWE-bench evaluates patches for repository issues (Jimenez et al., 2024). Their principal success measures do not distinguish the conciseness of accepted programs. The Google Code Golf Championship explicitly targets program length (Moffitt et al., 2025). The closest benchmark, CodeGolf Bench (Padwal, 2026), uses multilingual problems from `code.golf` and measures correctness and conciseness through pass@k and human-relative percentiles. ARC-Golf Bench builds on this direction by studying verifier-guided optimization over cumulative output-token budgets, behavioral trajectories, and experience transfer.

Program optimization has a long history as a search problem. Stochastic superoptimization searches for efficient low-level implementations (Schkufza et al., 2013); PIE studies performance-improving edits using human examples, retrieval, and model adaptation (Shypula et al., 2024); and AlphaEvolve combines language models with evolutionary search and automated evaluation (Novikov et al., 2025). Code golf provides a complementary setting with an exact program-length objective, avoiding runtime measurement noise while supporting sustained optimization and experience reuse.

Trajectory analysis and code similarity provide tools for understanding agent behavior. TRAJECT-Bench evaluates tool selection, arguments, and dependency ordering beyond final accuracy (He et al., 2025). CodeBLEU combines token, syntax-tree, and data-flow matching (Ren et al., 2020), while GraphCodeBERT incorporates data flow into representations evaluated on tasks including clone detection (Guo et al., 2021). Our analysis focuses on exploration among implementations of the same task. Golf-UnZip exposes local compression techniques through verified expansion and identifies broader methods through task-specific structural signatures. These interpretable measurements, together with verification behavior, complement outcome-based evaluation.

Experience abstraction enables reuse across tasks. ExpeL extracts natural-language insights (Zhao et al., 2024), and Voyager accumulates executable skills (Wang et al., 2024). Subsequent work shows that skill granularity and format affect transfer and can cause negative effects (Feng et al., 2026); SkillCommit constructs hierarchical skills through behavioral validation and evaluates cross-model transfer (He & Yang, 2026). We investigate abstraction under a program-length objective, where reusable tricks can be entangled with task-specific representations.

Skill transferability also depends on the model and harness. SkillOpt evaluates optimized skill documents across models and execution settings (Yang et al., 2026), MASA adapts skills to receiving models (Yu et al., 2026), and Skill-Use examines skill triggering, procedural compliance, and harness dependence (Han et al., 2026). SWE-agent highlights interface design (Yang et al., 2024), while Agentic Harness Engineering and HarnessDev study harness evolution and its transfer across models (Lin et al., 2026; Wu et al., 2026). We examine whether adapting a receiving worker's search organization makes transferred skills more useful. Specifically, we compare skill guidance with explicit solution-family assignments against matched generic parallel exploration, measuring their combined effect on compression.

3 ARC-GOLF BENCHMARK CONSTRUCTION

A useful benchmark for continual program optimization should distinguish the agents' ability while remaining affordable to evaluate repeatedly. We therefore select tasks spanning a range of optimization difficulties and prioritize those that discriminate among levels of solution quality.

3.1 PROBLEM SET SELECTION

Evaluating agents on all 400 competition tasks is costly. We therefore curate a compact set of 96 tasks that distinguishes levels of optimization ability while spanning a range of difficulties. To guide selection, we fit a graded response model (GRM) (Samejima, 1969) to final submissions of human teams, estimating each task's **discrimination** a_t and **difficulty thresholds** $b_{t,\ell}$.

Encoding relative solution quality. For each team–task submission, we divide its byte count by the shortest human solution for that task. This ratio measures compression quality relative to the task's best human reference. We pool these ratios together and discretize them using four shared quintile cutpoints, producing ordinal scores $Y_{h,t} \in \{1, \ldots, 5\}$ for team h on task t. Higher scores indicate better solutions. Appendix A provides the encoding details.

We next estimate the GRM parameters by marginal maximum likelihood using expectation–maximization (EM) (Dempster et al., 1977), following the Bock–Aitkin framework (Bock & Aitkin, 1981). The GRM relates these scores to a latent team ability θ_h:

$$\Pr(Y_{h,t} > \ell \mid \theta_h) = \sigma(a_t(\theta_h - b_{t,\ell})), \qquad \ell = 1, \ldots, 4,$$

where σ is the logistic function. Here, the key quantities are:

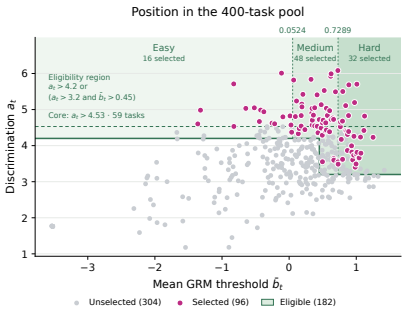


Figure 1: GRM difficulty and discrimination of 400 tasks. Selected tasks (magenta) comprise easy / medium / hard tasks.

- $Y_{h,t} \in \{1, \ldots, 5\}$: the score of team h on task t. Higher scores indicate better solutions;

- θ_h: the latent ability of team h. A larger θ_h indicates greater ability of team h;
- $a_t > 0$: the discrimination parameter of task t. A larger a_t indicates that the task better discriminates among teams;
- $b_{t,1} \le b_{t,2} \le b_{t,3} \le b_{t,4}$: ordered task difficulty thresholds, each indicating the ability needed for a 50% chance of scoring above the corresponding category.

We summarize task difficulty by the mean threshold, $\bar{b}_t = \frac{1}{4}\sum_{\ell=1}^{4} b_{t,\ell}$. The fitted GRM places the 400 tasks in a discrimination–difficulty plane (Figure 1).

Selecting tasks to construct ARC-Golf Benchmark. We include all 59 tasks with $a_t > 4.53$, then randomly sample 37 additional tasks from the remaining candidates satisfying

$$a_t > 4.2 \quad \text{or} \quad (a_t > 3.2 \text{ and } \bar{b}_t > 0.45).$$

This rule prioritizes highly discriminative tasks while also admitting harder tasks with moderately high discrimination. We sort the selected tasks by $\bar{b}_t$ and assign the first 16 to the easy group, the next 48 to medium, and the remaining 32 to hard. These labels describe relative difficulty within the selected benchmark. The resulting subset is concentrated toward the upper right of the pool, emphasizing high discrimination while including challenging tasks. Appendix A reports the estimation and selection details, and Table 4 lists all selected tasks.

3.2 Benchmark Environment Setup and Task Protocol

The benchmark harness follows three principles: isolated trials, controlled information access, and independent verification. Each trial starts in a fresh container from a version-pinned image, with read-only task specifications and an empty writable workspace. Agents receive task examples, the generator, verifier, and a reference byte target, but no reference solution. During verification, candidate programs cannot access the task data file.The verifier validates programs in accordance with the competition rules, the full details of which are provided in Appendix B.

Baseline trials use serial model calls under common output-token and wall-clock limits, with subagent and web-search tools disabled. The harness independently verifies candidates and tracks the shortest accepted program in UTF-8 bytes as output tokens accumulate. Verified candidates remain eligible for scoring after interruption, and execution traces are retained. Appendix C details the verification procedure, information controls and their limitations, resource limits, and recovery of saved candidates.

4 Evaluating Models and Harnesses on ARC-Golf Bench

We evaluate DeepSeek-V4-Flash, GLM-5.3-Flash, Qwen3.8-Flash, and GPT-5.6-Luna with Claude Code (v2.1.250) and Codex (v0.150.1), yielding 8 model–harness configurations. The four models span different providers, and the two general-purpose harnesses differ in tool use and execution control. Evaluating every model with both harnesses enables controlled comparisons of models within a harness and harnesses for a fixed model, without claiming an exhaustive ranking.

All configurations follow the basic task protocol in Section 3.2, using the same 96 tasks, task prompt, human-record targets, execution environment, and verification procedure. Model requests are serialized within each trial, and subagents and web tools are disabled. Each trial has a cumulative output-token limit of 3×10^6 and an 8-hour wall-clock limit; these are upper limits, and agents may terminate earlier. Reasoning effort is set to the maximum supported setting. These settings standardize the available controls without implying equal internal computation across models.

4.1 Trajectory and Outcome Metrics

We track the byte length of the shortest verified program found as output tokens accumulate. Let $L_{c,i,r}(b)$ denote this length for configuration c, task i, and trial r at cumulative output-token number b, with $L_{c,i,r}(b) = \infty$ if no candidate has passed verification.

At a certain number of output tokens b, we rank the best verified program against the recorded human solutions. Let $h_{i,j}$ be human team j's solution length and N_i the number of teams with recorded

solutions for task i. We define

$$R_{c,i,r}(b) = \begin{cases} 1 + \sum_{j=1}^{N_i} \mathbb{I}[h_{i,j} < L_{c,i,r}(b)], & L_{c,i,r}(b) < \infty, \\ N_i + 1, & \text{otherwise.} \end{cases} \tag{1}$$

We note that $\sum_{j=1}^{N_i} \mathbb{I}[h_{i,j} < L_{c,i,r}(b)]$ denotes the number of human teams with strictly shorter solutions: five shorter human solutions give rank 6. Rank 1 means matching or beating the best human solution. Before the first verified solution, the rank is $N_i + 1$. At each b, we average ranks over trials within each task and then over tasks.

4.2 Results

We average ranks over trials within each task and then over tasks, giving every task equal weight. Table 1 reports final mean ranks and accepted-run counts, overall and by difficulty. Figure 2 tracks the mean rank of the shortest verified solution against cumulative output tokens. Runs interrupted after finding a verified solution are scored using that solution, whereas runs that obtain no verified solution under the defined protocol are marked as **not pass** and have their ranks recorded as $N_i + 1$; case details are deferred to the Appendix D.

High acceptance leaves substantial room for optimization, especially on hard tasks. Verifier acceptance is high across all eight configurations, yet mean ranks range from 10.9 to 17.6, leaving even the best configuration well behind the strongest human solutions overall. Harder tasks further separate the systems: the gap between the best and worst configurations widens from 4.2 ranks on easy tasks to 8.6 on hard tasks. Qwen, DeepSeek, and GLM receive worse ranks as difficulty increases under both harnesses, whereas Codex·GPT maintains similar ranks (11.9, 12.0, and 11.3). Consequently, Codex·Qwen leads on easy and medium tasks, while Codex·GPT leads on hard tasks, highlighting differences in optimization capability beyond correctness.

Model performance depends on the harness. DeepSeek-V4-Flash, Qwen3.8-Flash, and GPT-5.6-Luna achieve better mean ranks under Codex, whereas GLM-5.3-Flash performs better under Claude Code. The harness also changes the model ordering: GLM leads under Claude Code with a mean rank of 14.2, but finishes last under Codex at 17.6; Qwen improves from 15.4 under Claude Code to 10.9 under Codex. Performance therefore reflects the interaction between the model and its execution environment, potentially influenced by the tools and interaction patterns represented in the model's training data.

Early progress does not determine final quality. The trajectories reveal differences that terminal scores conceal. At 100K output tokens, Codex·GLM ranks second among the eight configurations and Codex·Qwen sixth, yet they finish last and first, respectively. Conversely, Claude Code·GLM ranks last at 100K tokens but ultimately outperforms Claude Code·Qwen. Strong early performance and sustained improvement are distinct properties, motivating evaluation across the full trajectory.

5 Analyzing Agent Optimization Behavior

Performance trajectories show how agents improve their programs, but provide limited insight. We examine three complementary aspects of optimization behavior: the compression tricks used within individual solutions, the diversity of methods across solutions, and verification before Final.py update (a submit function call). We first describe how we identify tricks and solution methods, then define the measurements and analyze differences across models, harnesses, and task difficulties.

5.1 Identifying Compression Tricks and Solution Methods

Exposing compression tricks with Golf-Unzip. Highly compressed programs can obscure both local shortcuts and broader solution structure. Golf-Unzip uses an LLM judge (DeepSeek-V4.1-Flash) to expand one compression mechanism at a time (Figure 3 shows the complete mechanism

Table 1: Final mean relative ranks and verifier acceptance by configuration and task difficulty. Lower ranks are better; rank 1 matches or beats the best human solution, and failures receive $N_i + 1$.

	All		Easy		Medium		Hard	
Configuration	Rank	Pass	Rank	Pass	Rank	Pass	Rank	Pass
Claude Code·DeepSeek-V4-Flash	15.6	288/288	10.4	48/48	16.3	144/144	17.3	96/96
Claude Code·GLM-5.3-Flash	14.2	277/288	9.5	48/48	13.8	140/144	17.1	89/96
Claude Code·Qwen3.8-Flash	15.4	288/288	10.3	48/48	15.7	144/144	17.6	96/96
Claude Code·GPT-5.6-Luna	17.4	287/288	9.3	48/48	17.9	144/144	20.6	95/96
Codex·DeepSeek-V4-Flash	14.2	275/288	9.4	46/48	15.5	136/144	14.8	93/96
Codex·GLM-5.3-Flash	17.6	288/288	13.0	48/48	18.3	144/144	18.9	96/96
Codex·Qwen3.8-Flash	**10.9**	288/288	**9.1**	48/48	**11.2**	144/144	11.4	96/96
Codex·GPT-5.6-Luna	11.8	288/288	11.9	48/48	12.0	144/144	**11.3**	96/96

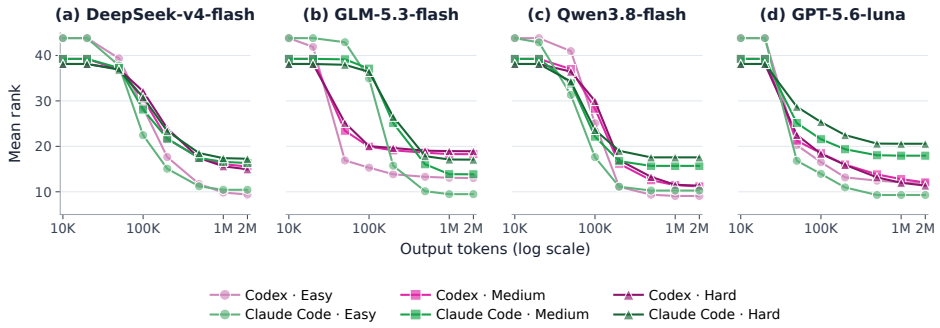


Figure 2: Mean human-relative rank of the shortest verified solution versus cumulative output tokens, shown on a logarithmic x-axis. Lower is better.

of golf-unzip with a finite-state machine). Each accepted replacement must pass the task's verification, recorded as an exposed trick. We retain completed expansion traces and use their normalized labels from a 39-family vocabulary constructed from pilot tasks. These labels describe mechanisms recovered through expansion, rather than reconstructing the agent's original reasoning process. Appendices E.1 and F.2 provide the procedure and vocabulary.

Indentifying solution methods with shared task rubrics. To distinguish broader solution approaches, the same judge constructs a task-specific rubric from expanded solutions across configurations. Each rubric specifies method axes and their possible labels; it is frozen before annotation and shared across configurations. Repeated annotation with voting assigns each solution a signature containing one label per axis. Identical complete signatures define the same method within this rubric. Appendix E.2 details the procedure and figure 4 pictures the specific process.

5.2 Measuring Trick Richness, Method Diversity, and Verification

Within-solution trick richness (TR). Trick Richness (TR) averages the number of distinct trick families per solution within a run, counting each family once (Appendix F.2). Higher TR indicates richer trick combinations within solutions.

Across-solution method diversity (MD). Method Diversity (MD) averages the fraction of differing method-axis labels over all solution pairs within a run. The score lies in $[0, 1]$: zero means all signatures agree, and higher values indicate greater method diversity. We also report the distinct method count K.

Verification rate before an update of Final.py (VR). VR is the run-averaged fraction of updates to `Final.py` matched to prior successful verification records. Higher VR suggests a more cautious submission practice, with successful verification more often recorded before updates to `Final.py`.

Both TR and MD metrics use the same verified solutions with completed Golf-Unzip expansions and method signatures. We retain runs with at least two such solutions, average their scores equally within each task, and then average equally across tasks. Harness comparisons use task/run pairs that meet these requirements in all eight configurations. Unsuccessful implementations are excluded. VR is averaged equally over runs with usable logs and at least one recorded update to `Final.py`. Appendix E.3 provides the detailed selection, aggregation, and verification-matching rules.

5.3 How Optimization Behavior Varies Across Models and Harnesses

Table 2 summarizes behavioral measurements for the eight configurations. We abbreviate models by series name, using the same versions as in Section 4. Appendix E.4– E.6 provides visualizations of difficulty trends in TR and MD, trick preferences, model-specific harness effects on behavior and performance, and overall behavioral profiles (a radar figure). Appendix F.3 includes more details about confidence intervals and statistical tests.

Table 2: Behavioral measurements by configuration. Trick Richness and Method Diversity measure trick richness and method diversity, averaging eligible runs within tasks and then tasks within settings. VR is the run-averaged fraction of final-file updates linked to prior successful verification.

	Codex				Claude Code			
Metric	Qwen	GPT	DeepSeek	GLM	Qwen	GPT	DeepSeek	GLM
TR	4.283	4.471	3.832	4.436	4.389	4.288	3.987	4.415
MD	0.290	0.359	0.309	0.289	0.231	0.261	0.325	0.246
VR	0.839	0.884	0.908	0.859	0.872	0.943	0.897	0.832

Common tricks recur across configurations and tasks. We measure each trick family's prevalence among retained solutions within each run, then average equally over runs, tasks, and settings. This *balanced frequency* prevents runs with more retained candidates from dominating. List comprehensions (81.0%) and lambdas (53.6%) are most frequent, and both appear on every task in the primary analysis. Appendix E.5 reports full frequencies and configuration-specific deviations.

Harness effects on optimization behavior depend on the model. In matched comparisons, Codex increases GPT's trick richness but decreases DeepSeek's, a difference of 0.455 families between the two effects (Figure 8a). It also increases method diversity for GPT, GLM, and Qwen, most strongly for GPT (Figure 8b). Codex increases generator-expression prevalence for GLM by 12.8 percentage points but decreases walrus prevalence for DeepSeek by 9.4 points (Figure 7); both effects remain significant after Benjamini–Hochberg correction (Benjamini & Hochberg, 1995). VR is higher under Claude Code for Qwen and GPT, but under Codex for DeepSeek and GLM.

Harder tasks involve more trick families. Averaged across settings, hard-task solutions contain 0.958 more trick families than easy-task solutions (Figure 6). Method diversity is lower on hard tasks. Hard tasks also receive more `other` labels, reducing the method rubrics' resolution. Thus, lower measured diversity describes the annotated method space and does not establish that agents attempt fewer approaches.

Behavioral measurements complement optimization performance. Trick Richness and Method Diversity describe distinct aspects of retained solutions: on matched data, their association is weak across setting averages and modestly negative after removing task and setting means (see Table 9, in Appendix F.3). Neither consistently favors the better-performing harness. Compared with Claude Code, Codex yields higher method diversity but worse performance for GLM, and lower trick richness but better performance for DeepSeek (Figure 8c). Claude Code with GPT has the highest VR (0.943), whereas Codex with Qwen achieves the best performance despite lower VR (0.839).

These measurements characterize optimization behavior without implying that greater richness, diversity, or verification linkage ensures better compression.

6 Transferring Experience Across Problems and Models

Having evaluated program optimization on individual problems, we next ask whether experience from previous problems can improve subsequent solutions. Code golf makes this transfer challenging: a compact program tightly couples its algorithm and syntax to properties of a particular task.

6.1 Constructing the Skill Library

A direct approach to experience reuse is to place related source problems and their solutions in the worker's context. Preliminary trials did not reveal consistent improvements, motivating a more structured representation of optimization knowledge. We therefore distill prior solutions into a structured skill library and examine its use on other problems. We call the model that extracts the skills the *distiller* and the model that applies them the *worker*.

We construct the library from 50 source tasks, from two reference solutions of each task: the best recorded agent solution and the best known human solution. Distillation focuses on the skill choices responsible for reducing program length and the conditions under which those choices can be reused.

Three levels of optimization skills. Prior work shows that the granularity of induced skills affects cross-task transfer (Feng et al., 2026), and explores hierarchical abstraction supported by behavioral validation (He & Yang, 2026). For code golf, we distinguish three levels according to the scope of the proposed change and organize them in a JSON file: **Syntactic simplifications** shorten local expressions or code fragments while preserving their behavior under the stated applicability conditions; **Conditional rewrites** replace a computation with a shorter alternative when specific properties of the inputs or task hold; **Representation strategies** describe broader algorithmic structures, the task properties that make them applicable, and measured skeleton costs where available, which guide the choice among alternative solution families.

Recognizing when to apply a skill. Effective skill reuse requires the *worker* to recognize relevant optimization opportunities in the current task or program. To support this, each skill includes two fields: *trigger*, which specifies when the skill may apply, and *eliminates*, which identifies the computation or code structure the skill can remove. Together, these fields connect applicability conditions to concrete opportunities for compression. The entire library is included in the worker's context.

6.2 Cross-Problem Transfer with a Skill Library

Setup and metrics. We compare performance with and without the skill library on ten held-out tasks, using two workers with substantially different capabilities: Claude Fable 5.1 (Worker A) and DeepSeek `v4-flash` (Worker B). This comparison tests whether the library benefits workers at different capability levels. Each task starts in a fresh session with no memory of other target tasks. Candidates are evaluated using the official verifier, and final solutions are independently re-verified.

We report total bytes and task-level win–tie–loss counts, comparing verified solutions with and without the library. Total bytes measure the aggregate magnitude of improvement, while win–tie–loss counts give each task equal weight.

Cross-problem transferability. For Worker A, the library reduces total program length from 1023 to 888 bytes, a saving of 13.5 bytes per task (13.2%), with seven wins, one tie, and two losses. These results indicate beneficial transfer in the observed runs, although the magnitude of the gains may vary across repeated runs. For Worker B, however, the library increases total length from 1120 to 1157 bytes (3.3%), with three wins, one tie, and six losses. This contrast is consistent with prior findings that skill effectiveness can depend on the receiving model (Yu et al., 2026). We investigate whether adapting Worker B's harness can improve its use of transferred skills.

Table 3: Comparison on ten tasks using the same library. Lower total verified bytes are better; wins/ties/losses compare library-enabled against library-free conditions. Worker A and unadapted Worker B use one session per task and condition. For Worker B (adapted harness), both conditions use three isolated sessions per task, and the shortest independently verified candidate is retained.

	Library-free	Library-enabled	Byte Change(%)	Win / tie / loss
Worker A	1023	888	-13.2	7 / 1 / 2
Worker B	1120	1157	+3.3	3 / 1 / 6
Worker B (adaptive harness)	1117	1012	-9.4	8 / 0 / 2

6.3 Cross-Model Transfer and Harness Adaptation

Worker B's negative transfer motivates adapting its harness. trajectories suggest that it can verify and compress programs within a solution family but struggles to explore structurally different alternatives. Library guidance may therefore reinforce commitment to a representation with limited compression potential. We address this by organizing exploration explicitly, consistent with evidence that skill use depends on the agentic harness (Han et al., 2026).

Our *lane-conditioned harness* assigns three parallel sessions to distinct solution families: candidate enumeration with scored selection, recursion or rotation, and direct construction or separable transformations. Each lane provides applicability conditions, a program skeleton, implementation notes, and prioritized skill entries, while retaining the full library in context. Sessions optimize their assigned family and may switch only after explaining its structural infeasibility. The same lane catalog applies to every task, and the shortest independently verified candidate is retained.

The library-free control uses three sessions prompted to explore a natural formulation, an alternative program structure, and an alternative data representation. Both conditions share session counts, budget limits, and verification procedures, with isolated sessions. This comparison measures the combined benefit of skill guidance and explicit family assignments over generic parallel exploration.

Worker B improves from 1117 to 1012 total bytes under best-of-three selection, saving 105 bytes (9.4%), with eight wins and two losses (Table 3). This reverses the negative transfer observed in the single-session setting. Separate lanes preserve alternative approaches, and verified selection retains the shortest successful candidate. The results suggest that structured harness design can improve the usefulness of transferred skills.

7 Conclusion and Limitation

We introduce ARC-Golf Bench, a compact, difficulty-stratified benchmark for continual program optimization. Its exact byte-length objective, executable verification, and human references provide a clean testbed for studying how agents improve solutions as computation accumulates. Evaluating four models with two harnesses reveals that optimization performance depends on their interaction, while Golf-UnZip characterizes the behavior behind these outcomes. Our transfer experiments further show that structured skills and harness adaptation can help agents reuse optimization experience. Together, the benchmark and analysis tools support systematic evaluation of both LLM capabilities and agentic harness design.

Limitations. Our evaluation covers a limited set of models and harnesses. In particular, inference costs prevent extensive evaluation of the largest and most computationally demanding models, so the results do not establish the performance ceiling of current systems. Code golf also prioritizes brevity, which has limited direct relevance to software quality and may conflict with readability or maintainability. Its controlled setting helps isolate optimization behavior, but the findings may not generalize to objectives such as runtime, memory consumption, or compilation efficiency. Future work should extend this evaluation approach to practical program-optimization tasks with reliable correctness checks and other measurable benefits.

AI USE STATEMENT

The authors developed the research ideas, task selection, choice of the GRM framework, Trick Richness and Method Diversity metrics, Golf-Unzip approach, and knowledge-transfer methodology. The observations, analyses, and scientific conclusions were also developed by the authors.

We used Codex and Claude Code to assist with experiment scripts, and AI tools to generate plotting code and assist with manuscript drafting, rephrasing, and language editing. The authors manually reviewed the plotting code and script outputs. Model-based program generation, expansion, and annotation form part of the experimental methodology and are described in the corresponding sections.

The authors take full responsibility for the final paper, including its text, claims, results, and AI-assisted artifacts.

ETHICS STATEMENT

We use tasks from the NeurIPS 2025 Google Code Golf Championship and performance records from 48 leading teams. The competition rules, Section 4(a) of the Competition-Specific Rules permit academic research use of Competition Data. We credit the original creators and human teams for the original data, and we will respect artifact-specific redistribution permissions.

REPRODUCIBILITY STATEMENT

The experimental setup and appendices describe task selection, evaluation protocols, human-reference calibration, Golf-Unzip, and statistical analyses. We will provide an anonymous package containing code, archived analysis inputs, dependencies, checksums, and third-party acquisition instructions. Recomputing archived analyses will be distinguished from rerunning model inference, which depends on API availability and sampling variability.

REFERENCES

Yoav Benjamini and Yosef Hochberg. Controlling the false discovery rate: A practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society: Series B (Methodological)*, 57(1):289–300, 1995. doi: 10.1111/j.2517-6161.1995.tb02031.x.

R. Darrell Bock and Murray Aitkin. Marginal maximum likelihood estimation of item parameters: Application of an EM algorithm. *Psychometrika*, 46(4):443–459, 1981. doi: 10.1007/BF02293801.

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

François Chollet. On the measure of intelligence. *arXiv preprint arXiv:1911.01547*, 2019.

A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society: Series B (Methodological)*, 39(1):1–22, 1977. doi: 10.1111/j.2517-6161.1977.tb01600.x.

Yiyang Feng, Biddut Sarker Bijoy, Niranjan Balasubramanian, and Jiawei Zhou. Break it down, pass it on: Cross-task skill transfer in LLM agents. *arXiv preprint arXiv:2608.20274*, 2026.

Google LLC. NeurIPS 2025 - Google code golf championship: Official competition rules, 2025. URL https://www.kaggle.com/competitions/google-code-golf-2025/rules. Competition rules; accessed September 25, 2026.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. GraphCodeBERT: Pre-training code representations with data flow. In *International Conference on Learning Representations*, 2021.

Jinyi Han, Yuanjian Xu, Ying Liao, Xinyi Wang, Zishang Jiang, Zixiang Di, Fanyang Lu, Zhichao Hu, and Yanghua Xiao. Skill-Use: Can LLMs actually use skills in agentic harnesses? *arXiv preprint arXiv:2608.04828*, 2026.

Pengfei He, Zhenwei Dai, Bing He, Hui Liu, Xianfeng Tang, Hanqing Lu, Juanhui Li, Jiayuan Ding, Subhabrata Mukherjee, Suhang Wang, Yue Xing, Jiliang Tang, and Benoit Dumoulin. TRAJECT-Bench: A trajectory-aware benchmark for evaluating agentic tool use. *arXiv preprint arXiv:2510.04550*, 2025.

Yu He and Weikai Yang. SkillCommit: Evolving agent skills through behaviorally validated scope expansion. *arXiv preprint arXiv:2608.15165*, 2026.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. In *International Conference on Learning Representations*, 2025.

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*, 2024.

Jiahang Lin, Shichun Liu, Chengjun Pan, Lizhi Lin, Shihan Dou, Xuanjing Huang, Hang Yan, Zhenhua Han, and Tao Gui. Agentic harness engineering: Observability-driven automatic evolution of coding-agent harnesses. *arXiv preprint arXiv:2604.25850*, 2026.

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.

Michael D. Moffitt, Walter Reade, Divy Thakkar, Ryan Burnell, Addison Howard, and Orhan Firat. The NeurIPS 2025 Google Code Golf Championship. Official competition website, 2025. URL https://sites.google.com/view/neurips-2025-code-golf. Competition organizers; accessed September 10, 2026.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

Vedant Padwal. CodeGolf Bench: A multi-language benchmark for evaluating concise code generation capabilities of large language models. *arXiv preprint arXiv:2605.30394*, 2026.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. CodeBLEU: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020.

Fumiko Samejima. *Estimation of Latent Ability Using a Response Pattern of Graded Scores*. Psychometric Monograph No. 17. Psychometric Society, Richmond, VA, 1969. doi: 10.1007/BF03372160. URL https://www.psychometricsociety.org/sites/main/files/file-attachments/mn17.pdf.

Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, Yiming Yang, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. Learning performance-improving code edits. In *International Conference on Learning Representations*, 2024.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research*, 2024.

Yuhao Wu, Jingyuan Zhang, Jiajun Shi, Xinping Lei, Qingshui Gu, Yuxuan Zhang, Zexuan Wang, Chen He, Chen Huang, Maojia Song, Zhiyuan Zeng, Shaowen Wang, Jinkai Liu, Yunfeng Shi, Jiaheng Liu, Shen Yan, Wenhao Huang, Ge Zhang, and Wenxuan Zhang. HarnessDev: Can LLMs create and evolve their own agent harness? *arXiv preprint arXiv:2609.01437*, 2026.

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

Yifan Yang, Ziyang Gong, Weiquan Huang, Qihao Yang, Ziwei Zhou, Zisu Huang, Yan Li, Xuemei Gao, Qi Dai, Bei Liu, Kai Qiu, Yuqing Yang, Dongdong Chen, Xue Yang, and Chong Luo. SkillOpt: Executive strategy for self-evolving agent skills. *arXiv preprint arXiv:2605.23904*, 2026.

Jianxiang Yu, Jiapeng Zhu, Bochen Lin, Qier Cui, Zichen Ding, and Xiang Li. Skill is not one-size-fits-all: Model-aware skill alignment for LLM agents. *arXiv preprint arXiv:2605.30723*, 2026.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19632–19642, 2024.

Appendix

A GRM Implementation Details

Input. A byte matrix $X \in \mathbb{R}^{400\times 48}$, where $X_{t,h}$ denotes the best submission byte count of human team h on task t. The input matrix retains its task-by-team layout: 400 task rows and 48 team columns. Ordinal responses below use respondent-first notation $Y_{h,t}$; this notation does not change the layout of the input matrix. Missing cells are treated as missing at random.

A.1 Step 1: Normalization and Record Ratios

For each task t, compute the in-sample record

$$\mathrm{rec}_t = \min_h X_{t,h}. \tag{2}$$

For each observed cell (t, h), define the record ratio

$$\rho_{t,h} = \frac{X_{t,h}}{\mathrm{rec}_t} \geq 1. \tag{3}$$

A.2 Step 2: Ordinal Response Binning

Take the pooled quantiles $P_{80}, P_{60}, P_{40}, P_{20}$ of all observed record ratios as four cutpoints

$$c_1 = P_{80}, \quad c_2 = P_{60}, \quad c_3 = P_{40}, \quad c_4 = P_{20}, \tag{4}$$

satisfying $c_1 > c_2 > c_3 > c_4$. Map each observation to a five-category ordinal response:

$$Y_{h,t} = 1 + \sum_{k=1}^{4} \mathbb{I}(\rho_{t,h} \leq c_k) \in \{1, \ldots, 5\}. \tag{5}$$

Thus $Y = 1$ denotes the worst performance, and $Y = 5$ denotes the closest to the record, where $\mathbb{I}(\cdot)$ is the indicator function taking value 1 if its argument is true and 0 otherwise, so that $\mathbb{I}(\rho_{t,h} \leq c_k)$ indicates whether $\rho_{t,h}$ falls at or below the threshold c_k.

A.3 Step 3: Graded Response Model and Likelihood

We use the Samejima graded response model (GRM). Each human team h has an ability value θ_h. For task t, let the discrimination be $a_t > 0$, with ordered thresholds

$$b_{t,1} < b_{t,2} < b_{t,3} < b_{t,4}. \tag{6}$$

Write $b_t = (b_{t,1}, b_{t,2}, b_{t,3}, b_{t,4})$ for the threshold vector. The cumulative probabilities are

$$P_{t,k}(\theta_h) = \Pr(Y_{h,t} \geq k + 1 \mid \theta_h) = \sigma\big(a_t(\theta_h - b_{t,k})\big), \quad k = 1, \ldots, 4, \tag{7}$$

where σ is the logistic function. The argument θ_h makes the dependence on the human team's ability explicit. Category probabilities are given by differences between adjacent cumulative probabilities:

$$\Pr(Y_{h,t} = 1 \mid \theta_h) = 1 - P_{t,1}(\theta_h), \tag{8}$$

$$\Pr(Y_{h,t} = k \mid \theta_h) = P_{t,k-1}(\theta_h) - P_{t,k}(\theta_h), \quad k = 2, 3, 4, \tag{9}$$

$$\Pr(Y_{h,t} = 5 \mid \theta_h) = P_{t,4}(\theta_h). \tag{10}$$

A.4 Step 4: EM Estimation

We use the Bock–Aitkin marginal maximum likelihood EM algorithm. The ability prior is standard normal $N(0, 1)$, and integration uses a 41-point Gauss–Hermite grid $\{\theta_g\}$. We initialize $a_t = 1$ and $b_t = (-1, -0.3, 0.3, 1)$. We iterate the following two steps until convergence.

E-step. Fix the task parameters (a, b). For each human team h, compute on the grid the unnormalized posterior weights

$$\tilde{W}_{h,g} = q_g \prod_{t \in \mathcal{O}_h} \Pr\big(Y_{h,t} \mid \theta_g, a_t, b_t\big), \tag{11}$$

where q_g are the normalized Gauss–Hermite quadrature weights for integration under the standard normal prior, and $\mathcal{O}_h$ is the set of observed tasks for human team h. Then normalize

$$W_{h,g} = \frac{\tilde{W}_{h,g}}{\sum_{g'} \tilde{W}_{h,g'}}, \tag{12}$$

obtaining the posterior weight matrix $W \in \mathbb{R}^{48\times 41}$, with human teams indexing rows and grid nodes indexing columns.

M-step. Fix W, and maximize task by task the expected complete-data log-likelihood

$$Q_t(a_t, b_t) = \sum_h \sum_g W_{h,g} \log \Pr\big(Y_{h,t} \mid \theta_g, a_t, b_t\big), \tag{13}$$

under the constraints $a_t \in [0.05, 12]$ and $b_{t,1} < b_{t,2} < b_{t,3} < b_{t,4}$, using golden-section coordinate ascent to update (a_t, b_t).

A.5 Step 5: Ability Estimation and Standardization

After convergence, each human team's ability is taken as the posterior mean (EAP):

$$\hat{\theta}_h = \sum_g W_{h,g} \theta_g. \tag{14}$$

We then standardize $\hat{\theta}$ to mean 0 and standard deviation 1, and transform the task parameters accordingly to keep the model unchanged:

$$b_{t,k} \mapsto \frac{b_{t,k} - \mu_{\text{human}}}{\sigma_{\text{human}}}, \qquad a_t \mapsto a_t \, \sigma_{\text{human}}, \tag{15}$$

where μ_{human} and σ_{human} are the mean and standard deviation of the human ability estimates $\hat{\theta}_h$ before standardization. The final output consists of $(\hat{a}_t, \hat{b}_{t,1}, \ldots, \hat{b}_{t,4})$ for each task and $\hat{\theta}_h$ for each human team.

A.6 Step 6: Model Scoring and Human-Equivalent Rank

After the human calibration in Steps 1–5, we fix the standardized task parameters $(\hat{a}_t, \hat{b}_{t,1}, \ldots, \hat{b}_{t,4})$ for the 96 anchor tasks. For setting s, let $X_{s,t}$ be the byte count of its shortest available verified program on task t, and let rec_t be the minimum submission length among the 48 reference human teams. We encode $\rho_{s,t} = X_{s,t}/\mathrm{rec}_t$ into the same five-category ordinal response $Y_{s,t}$ using the cutpoints from Step 2. Here setting-level byte counts and responses use (s, t) indexing; the human input matrix in Step 1 retains its task-by-team indexing $X_{t,h}$. The observed set $\mathcal{O}_s$ contains tasks with an available verified solution; tasks without one are omitted from the likelihood.

For fixed-parameter model scoring, we use the uniform grid $\theta_g = -6 + 0.005g$, $g = 0, \ldots, 2400$, and a standard normal prior. This numerical integration is separate from the Gauss–Hermite quadrature used for human calibration. The normalized posterior weights and EAP estimate are

$$\widetilde{W}_{s,g} = \exp(-\theta_g^2/2) \prod_{t \in \mathcal{O}_s} \Pr(Y_{s,t} \mid \theta_g, \hat{a}_t, \hat{b}_t), \tag{16}$$

$$W_{s,g} = \frac{\widetilde{W}_{s,g}}{\sum_{g'} \widetilde{W}_{s,g'}}, \qquad \hat{\theta}_s = \sum_g W_{s,g} \theta_g. \tag{17}$$

The equal grid spacing and normal-density constant cancel in normalization. The human task parameters and reference ability estimates are not re-estimated.

We report the setting's insertion position among all $H = 48$ reference human teams:

$$\text{GRM-Rank}_s = 1 + \sum_{h \in \mathcal{H}} \mathbb{I}\{\hat{\theta}_h > \hat{\theta}_s\}. \tag{18}$$

Lower is better; the range is 1 to 49. The strict inequality places an exact tie at the best occupied position of that tie group. Each setting is inserted independently, so two settings may share a rank despite having different EAP estimates. These ranks are descriptive point estimates conditional on the fixed human calibration and the available verified responses.

Ratios below one fall in the highest ordinal category, $Y = 5$, which also includes some ratios above one under the calibrated cutpoints. The highest category therefore does not isolate strict improvements over the human reference. We separately report *Records*, the number of tasks on which a setting's verified byte count is strictly smaller than the minimum in the 48-team human reference.

A.7 Full Benchmark Selection

Table 4 lists all 96 tasks, numbered consecutively in table order, and their source task IDs in the original 400-task pool. The difficulty summary is $\bar{b}_t = \frac{1}{4}\sum_{\ell=1}^{4} b_{t,\ell}$. Using the stored parameters before rounding, easy tasks satisfy $\bar{b}_t < 0.052375$, medium tasks $0.052375 \le \bar{b}_t < 0.728875$, and hard tasks $\bar{b}_t \ge 0.728875$, giving 16, 48, and 32 tasks, respectively.

Table 4: Complete 96-task benchmark selection. Groups follow the 16/48/32 partition by increasing $\bar{b}_t$; GRM parameters come from the published calibration. Values are shown to three decimal places.

Task	Source task	Group	a_t	$b_{t,1}$	$b_{t,2}$	$b_{t,3}$	$b_{t,4}$	$\bar{b}_t$
task1	task256	Easy	6.010	-2.050	-0.375	0.702	1.268	-0.114
task2	task329	Easy	5.711	-2.067	-0.711	-0.326	-0.199	-0.826
task3	task304	Easy	5.048	-0.862	-0.755	-0.495	0.038	-0.518
task4	task060	Easy	5.033	-1.552	-1.326	-0.351	0.647	-0.646
task5	task217	Easy	4.986	-2.117	-0.574	-0.022	1.015	-0.424
task6	task315	Easy	4.983	-2.476	-2.103	-0.486	-0.218	-1.321
task7	task239	Easy	4.939	-2.121	-0.632	-0.043	1.290	-0.376
task8	task290	Easy	4.824	-1.164	-0.364	-0.070	1.652	0.014
task9	task149	Easy	4.798	-0.912	-0.805	-0.308	1.653	-0.093
task10	task318	Easy	4.669	-2.152	-1.164	0.094	1.449	-0.443
task11	task267	Easy	4.624	-2.150	-0.482	0.646	0.967	-0.255
task12	task303	Easy	4.605	-0.793	-0.457	-0.110	0.496	-0.216
task13	task211	Easy	4.602	-2.508	-2.136	-0.538	-0.262	-1.361
task14	task236	Easy	4.572	-0.897	-0.739	0.288	1.459	0.028
task15	task045	Easy	4.532	-1.428	-1.221	-0.627	-0.017	-0.823
task16	task177	Easy	4.372	-0.672	-0.445	-0.057	1.324	0.038
task17	task169	Medium	6.082	-0.417	0.522	1.143	1.656	0.726
task18	task277	Medium	5.991	-0.706	0.415	1.138	1.654	0.625
task19	task189	Medium	5.923	-0.738	-0.222	1.261	2.176	0.619
task20	task354	Medium	5.838	-0.393	-0.005	0.538	1.165	0.326
task21	task214	Medium	5.818	-0.867	-0.528	0.623	1.041	0.067
task22	task019	Medium	5.528	-0.707	0.200	0.944	1.653	0.522
task23	task025	Medium	5.492	-1.134	0.519	1.195	2.182	0.690
task24	task384	Medium	5.422	-0.646	-0.364	0.545	1.286	0.205
task25	task092	Medium	5.408	-0.839	0.135	0.721	1.648	0.416
task26	task313	Medium	5.237	-0.654	-0.403	0.316	1.182	0.110
task27	task331	Medium	5.151	-0.705	0.097	0.432	0.933	0.189
task28	task168	Medium	5.135	-0.456	0.388	0.686	1.294	0.478
task29	task021	Medium	5.074	-1.316	-0.196	0.938	1.338	0.191
task30	task046	Medium	5.049	-1.248	0.123	0.980	1.668	0.381
task31	task369	Medium	5.035	-0.546	0.309	1.019	1.439	0.555
task32	task350	Medium	5.031	-0.569	-0.098	0.388	1.644	0.341
task33	task112	Medium	4.969	-1.138	0.494	1.118	1.940	0.604
task34	task344	Medium	4.906	-0.297	0.082	0.272	1.296	0.338
task35	task037	Medium	4.845	-0.693	0.028	0.670	1.443	0.362
task36	task260	Medium	4.836	-0.776	0.314	1.070	1.670	0.570
task37	task246	Medium	4.833	-1.765	-0.015	1.013	1.487	0.180
task38	task270	Medium	4.806	-1.367	0.161	0.540	1.039	0.093
task39	task254	Medium	4.765	-0.462	0.025	1.103	1.941	0.652
task40	task204	Medium	4.727	-0.282	0.616	0.858	1.457	0.662
task41	task184	Medium	4.727	-0.256	0.318	0.966	1.473	0.625
task42	task088	Medium	4.722	-0.553	0.064	0.808	1.479	0.450
task43	task013	Medium	4.705	-1.146	0.123	1.110	1.943	0.508
task44	task124	Medium	4.705	-0.507	-0.030	0.469	1.450	0.346
task45	task190	Medium	4.632	-0.165	0.291	0.724	1.332	0.546
task46	task095	Medium	4.631	-0.999	0.021	1.029	1.936	0.497
task47	task259	Medium	4.614	-0.537	0.296	0.744	0.864	0.342

Continued on next page

Table 4 (continued)

Task	Source task	Group	a_t	$b_{t,1}$	$b_{t,2}$	$b_{t,3}$	$b_{t,4}$	$\bar{b}_t$
task48	task271	Medium	4.569	-0.755	-0.166	0.626	1.664	0.342
task49	task093	Medium	4.547	-0.407	0.183	0.623	1.123	0.380
task50	task237	Medium	4.542	-0.477	-0.209	0.459	0.944	0.179
task51	task365	Medium	4.536	-0.859	0.072	0.787	1.675	0.419
task52	task051	Medium	4.488	-0.507	0.224	0.861	1.328	0.476
task53	task128	Medium	4.475	-0.505	-0.232	0.294	1.044	0.150
task54	task104	Medium	4.443	-1.154	-0.018	0.636	1.476	0.235
task55	task055	Medium	4.433	-0.529	0.110	0.955	1.685	0.555
task56	task090	Medium	4.411	-1.602	0.096	1.632	2.026	0.538
task57	task306	Medium	4.384	-0.506	0.436	1.102	1.377	0.602
task58	task201	Medium	4.357	-1.786	0.087	1.439	1.979	0.430
task59	task381	Medium	4.330	-0.772	-0.612	0.256	1.674	0.136
task60	task340	Medium	4.289	-1.354	0.296	1.427	1.607	0.494
task61	task070	Medium	3.641	-0.709	0.419	1.208	1.739	0.664
task62	task302	Medium	3.597	-0.486	0.418	1.256	1.474	0.666
task63	task383	Medium	3.553	-0.408	0.264	0.744	1.386	0.496
task64	task159	Medium	3.550	-0.326	0.703	1.117	1.417	0.728
task65	task212	Hard	5.704	-0.161	0.339	1.645	2.213	1.009
task66	task004	Hard	5.685	-0.129	0.460	1.274	1.953	0.890
task67	task333	Hard	5.501	-0.526	0.258	1.281	2.184	0.799
task68	task145	Hard	5.194	-0.511	0.516	1.662	2.232	0.975
task69	task122	Hard	5.109	-0.274	0.512	1.188	1.944	0.842
task70	task086	Hard	4.833	-0.642	0.522	1.637	2.023	0.885
task71	task283	Hard	4.819	-0.134	0.702	1.678	2.250	1.124
task72	task120	Hard	4.680	-0.721	-0.030	1.676	2.255	0.795
task73	task250	Hard	4.653	-0.500	0.517	1.110	1.938	0.766
task74	task220	Hard	4.524	-0.452	0.179	1.123	2.203	0.763
task75	task222	Hard	4.420	0.176	0.567	1.480	2.233	1.114
task76	task132	Hard	4.417	-0.041	0.566	1.303	1.555	0.846
task77	task310	Hard	4.395	-0.346	0.236	1.336	2.219	0.861
task78	task125	Hard	4.391	-0.271	0.520	1.092	2.206	0.887
task79	task349	Hard	4.310	-0.773	0.420	1.650	2.182	0.870
task80	task034	Hard	4.259	-0.282	0.783	1.147	2.215	0.966
task81	task346	Hard	4.237	0.173	0.625	1.235	2.221	1.064
task82	task272	Hard	4.231	-0.241	0.708	2.155	2.380	1.250
task83	task338	Hard	4.174	-0.027	0.588	1.131	1.416	0.777
task84	task147	Hard	3.994	-0.388	0.358	1.493	2.257	0.930
task85	task044	Hard	3.865	-0.428	0.412	1.483	2.006	0.868
task86	task110	Hard	3.820	0.274	0.808	1.216	1.441	0.935
task87	task364	Hard	3.815	-0.252	0.964	1.348	1.988	1.012
task88	task153	Hard	3.791	0.068	0.739	1.445	2.026	1.070
task89	task336	Hard	3.774	0.058	0.485	1.610	1.838	0.998
task90	task036	Hard	3.737	-0.004	0.736	1.323	1.793	0.962
task91	task017	Hard	3.660	-0.568	0.703	1.710	2.325	1.042
task92	task198	Hard	3.620	-0.148	0.551	1.047	1.735	0.796
task93	task205	Hard	3.603	-0.393	0.702	1.472	2.009	0.948
task94	task185	Hard	3.491	-0.259	0.704	1.511	2.032	0.997
task95	task042	Hard	3.486	-0.191	0.605	1.099	1.407	0.730
task96	task317	Hard	3.403	-0.535	0.485	1.881	2.105	0.984

B Rules Governing a Code Golf Solution

The full competition rules for the NeurIPS 2025 Google Code Golf Championship are available at `https://www.kaggle.com/competitions/google-code-golf-2025/rules` (Google LLC, 2025). This appendix restates only the subset that bears directly on running

a task: what a solution must be, what it may import, what it may not do, and how compliance is decided.

B.1 THE SOLUTION

- **Form.** One Python 3 file defining an executable object named `p`. The official environment is Python 3.11 or newer, so 3.11-only syntax is legal.
- **Function.** `p` must map each input grid to its output grid, with every cell taking its expected value.
- **Metric.** The score is the file's size in *UTF-8 bytes*, not characters.
- **Fully materialized output.** The returned container and every one of its rows must be a `list` or a `tuple`. `map`, generator, filter and iterator rows are rejected: a lazily built grid does not count as a grid. The result is normalized to a list of lists and compared with ==, so values that compare equal to the expected integers (booleans, floats) are accepted.
- **Encoding.** The file must be valid UTF-8. Source whose bytes do not decode as UTF-8 is rejected before any other check runs.

B.2 IMPORTS

- **Any Python standard-library module may be imported**—`re`, `math`, `os`, `sys`, `itertools`, and the rest. Importing `os` or `sys` is thus permitted; what is forbidden is file access through them (below).
- **Third-party packages are forbidden**—`numpy`, `requests` and the like. The ban covers static imports (`import pkg`, `from pkg import ...`) and the dynamic routes `__import__('pkg')` and `importlib.import_module('pkg')`. The same rule holds at runtime, where an import attempted from inside the candidate is refused with an explicit error.
- **`eval` and `exec` are not restricted**: they are a legitimate compression device, and only file access is policed.

B.3 PROHIBITED

- **Reading the task data file.** `p` must be a pure function of its input.
- **Deriving outputs from the evaluated records.** The task-side consequence is that the output must be computed from the input.
- **Interactive and debugging entry points.** `input()` and `breakpoint()` are disabled.

C DETAILS OF ENVIRONMENT SETUP AND TASK PROTOCOL

We evaluate each task using a containerized execution harness released with the paper. The harness provides isolated task environments, consistent resource limits, and independent verification, while recording progress regardless of how a trial terminates.

Containerized Task Environment. Every trial, i.e., each repetition of a task, starts in a fresh Docker container built from the same version-pinned image. Task data, the generator and its utility library, the verifier, and the task guide are mounted read-only and constitute the permitted task specification. The generator is deliberately exposed as the specification of the intended transformation. Only `workdir/` is writable, and it is initially empty.

The agent receives the task guide and a reference byte target, but not the reference implementation. It must produce `workdir/Final.py`, defining a callable `p` that maps an input grid to an output grid. Verification checks both the generator's reference pairs and a hold-out split. A candidate is accepted only if it passes both phases, and its length is measured in UTF-8 bytes. The agent may invoke the verifier during optimization; the harness independently verifies candidates for scoring.

Information Control. During verification, candidate programs execute separately without access to the task data file, with data-file reads and non-standard imports blocked. Verification aborts if task

data are missing, writable, or incomplete. All model requests pass through a host-side proxy that removes subagent and web-search tools. An IPv4 firewall restricts outbound traffic, allowing bridge, DNS, local, and established connections. This is not complete network isolation: DNS remains an external channel, and firewall failures do not abort execution. Task rules prohibit deriving outputs from evaluation records.

Bounded Execution. Model requests are serialized within each trial, while independent trials may run concurrently. All configurations use the same per-call output limit, cumulative output-token budget, and wall-clock limit. The proxy records token usage and blocks further model calls after issuing a budget-exhaustion notification. At fixed output-token intervals, the harness verifies a capped number of saved candidates and records the shortest passing program observed so far.

After the agent terminates, but before container removal, the harness verifies `Final.py`. If it is missing or invalid, the harness checks saved candidates and marks recovered results as *salvaged*, recording their source filenames. Failed trials are not automatically rerun. Conversations, tool calls, token usage, verification outcomes, termination status, progress curves, and configuration metadata are retained. These records support pass rate, mean byte count among passing trials, and acceptance rates at specified reference byte targets.

D Failure Analysis

Twenty-five of the 2304 runs ended without an accepted solution.

- Twenty-four were attributable to the endpoint and the harness:
  - Thirteen runs under Codex·DeepSeek-V4-Flash ended before producing an accepted solution: eleven on responses truncated at the 128k per-response output limit and two on empty responses.
  - Eleven runs under Claude Code·GLM-5.3-Flash, spanning ten distinct tasks, ended on API errors: eight request timeouts and three responses exceeding the 128k output limit.
- The remaining run, under Claude Code·GPT-5.6-Luna on task 85 (run 2), completed without a recorded endpoint or harness fault but produced no accepted solution—the only capability failure observed in the grid.

E Details of the Behavioral Analysis

E.1 Golf-Unzip

Verified expansion. Golf-Unzip uses DeepSeek-V4.1-Flash to expose one compression mechanism per proposal, following the prompt in Appendix F.1. Each proposal includes a free-text mechanism label and a coarse trick code; the finer families used by Trick Richness are defined in Appendix F.2.

The source solution must first pass the task's verification cases. The runner then maintains the last accepted program in a bounded proposal–verification loop. A **STEP** proposes a replacement: passing verification archives the replacement and its metadata, resets the retry counter, and continues the loop. A rejected proposal leaves the accepted program unchanged and consumes a retry. A **DONE** declaration is accepted only when the scan for supported residual compression constructs is empty; otherwise it consumes a retry. A **BLUR** declaration, or exhaustion of the step or retry budget, terminates the trace as incomplete. API failures terminate as **ERROR**.

Each accepted expansion passes the configured task cases; this does not establish equivalence on every possible input. Completion depends on the judge and the supported residual scan. Only DONE traces enter the TR and MD analysis, with each solution's trick-family set taken from its full accepted expansion path. Archiving an expansion is separate from the agent's original updates to `Final.py`, which are measured by VR.

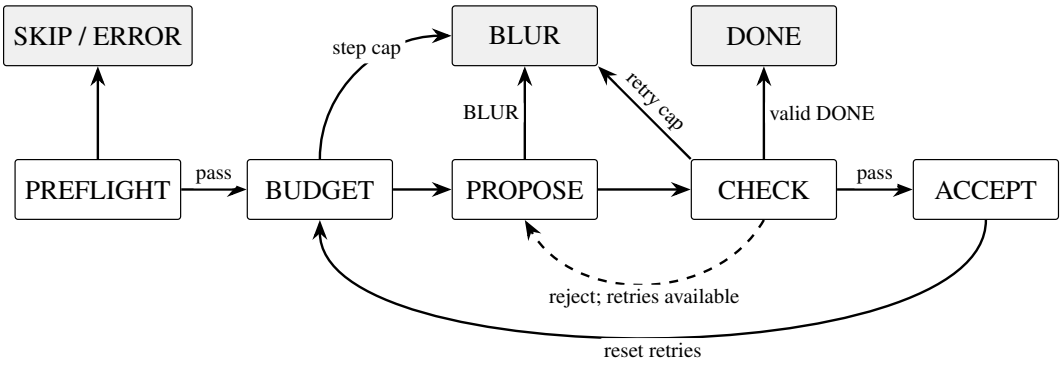


Figure 3: Golf-Unzip loop. CHECK validates completion or verifies a proposed replacement. Rejected proposals retry only within budget; BLUR is terminal. API failures terminate as ERROR (omitted from the diagram).

E.2 Method Recognition

Frozen method rubric. For each task t, the judge reads expanded solutions from multiple configurations and proposes a rubric with A_t method axes. Example axes include *construct*, *locate*, *extent*, and *rowbuild*; the *construct* vocabulary might contain $\{\text{rebuild}, \text{in-place}, \text{copy-prefix}\}$. Following verification and calibration, the rubric is frozen and used for every configuration and run on that task (Figure 4).

The annotation procedure defaults to $k = 3$ rounds per solution, with labels for all method axes requested in a single response per round. Request or parsing failures may be retried within a round; a round that yields no usable parsed response contributes no ballot. We aggregate labels separately for each axis by plurality vote: a strict majority is not required. Ties are resolved by selecting the tied label that first appears in vote order. No additional ballots are requested to resolve ties.

For a given axis, ballots without that axis are omitted from its tally. If no ballot supplies a label for the axis, its label is set to `other`. If the selected label is outside the axis vocabulary, it is also mapped to `other`. If all k rounds yield no usable ballots for a solution, the task-level annotation run terminates with `ERROR`.

Within a configuration–task–run combination (s, t, r), let i index solution records. The resulting labels form the signature

$$\sigma_i = (\sigma_{i,1}, \ldots, \sigma_{i,A_t}).$$

Two solutions belong to the same method group exactly when their complete signatures agree. This equality rule partitions the annotated records into method groups; distances between signatures provide the basis for Method Diversity.

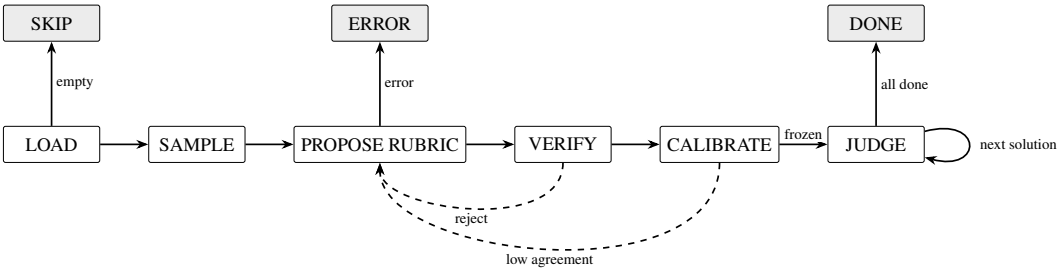


Figure 4: Method annotation with rubric verification, calibration, and voting on each axis.

E.3 METRIC DEFINITIONS AND AGGREGATION

GRM-Rank GRM-Rank reports a setting's human-equivalent rank: its insertion position among the reference set $\mathcal{H}$ of 48 human teams on a common graded response model (GRM) scale. For each task, the shortest verified program across available runs supplies the setting's byte count. We encode its ratio to the task's shortest human submission using the fixed calibration cutpoints. Tasks without a verified solution are omitted from the likelihood.

Let $\hat{\theta}_s$ be setting s's estimated ability on this fixed scale and $\hat{\theta}_h$ the corresponding estimate for human team h. We count how many human teams have higher estimated ability:

$$\text{GRM-Rank}_s = 1 + \sum_{h \in \mathcal{H}} \mathbb{I}\{\hat{\theta}_h > \hat{\theta}_s\}.$$

Lower ranks are better, with values from 1 to 49. Each setting is inserted independently; exact ties share the best occupied position. Ability estimation and numerical integration are detailed in Appendix A.6. These ranks are descriptive point estimates conditional on the fixed human calibration.

Within-solution trick richness (TR). Trick Richness measures the number of distinct trick families recovered from a typical completed solution's expansion trace. The 39-family vocabulary was constructed from 36 pilot tasks (Table 7). For setting s on task t in run r, $C_{(s,t,r)}$ denotes the average number of distinct trick families per eligible solution. Let $n_{(s,t,r)}$ be the number of eligible solutions in this combination, and let $F^{(i)}_{(s,t,r)}$ be the set of stored, non-noise family labels recovered from solution i.

$$C_{(s,t,r)} = \frac{1}{n_{(s,t,r)}} \sum_{i=1}^{n_{(s,t,r)}} \left|F^{(i)}_{(s,t,r)}\right|.$$

We then average eligible runs within each task and average tasks within each setting. Let $R_{(s,t)}$ be the eligible runs for setting s on task t, and let T_s be the eligible task set. The setting-level score is

$$\text{TR}_s = \frac{1}{|T_s|} \sum_{t \in T_s} \frac{1}{|R_{(s,t)}|} \sum_{r \in R_{(s,t)}} C_{(s,t,r)}.$$

A family contributes once per solution, regardless of how many accepted expansion steps use it. The score measures the richness recovered along the expansion path. We retain completed traces.

Across-solution method diversity (MD) For each task, a fixed rubric assigns each solution a method signature σ_i, a tuple of labels on A_t method axes. We measure diversity by the mean normalized Hamming distance between distinct solution pairs within a task/setting/run:

$$d_t(\sigma_i, \sigma_j) = \frac{1}{A_t} \sum_{a=1}^{A_t} \mathbb{I}\{\sigma_{i,a} \neq \sigma_{j,a}\}, \qquad V_{(s,t,r)} = \binom{n_{(s,t,r)}}{2}^{-1} \sum_{i<j} d_t(\sigma_i, \sigma_j).$$

Method Diversity averages $V_{(s,t,r)}$ over eligible runs within each task and then over tasks, as in Trick Richness. The score lies in $[0, 1]$, weights signatures by their observed frequencies, and excludes singleton runs. We also retain K, the number of distinct complete method signatures, and K/n, the distinct-method fraction among eligible solution records. The finite-sample relation to Rao's quadratic expression and the five-solution rarefaction summary are detailed in Appendix F.3. All method comparisons remain conditional on the task-specific rubrics, including the retained `other` category.

Verify-Rate (VR) VR measures how often updates to `Final.py` are matched to successful verification records in the agent's logs (see figure 5), capturing the extent to which the agent verifies solutions before submitting them. For setting s on task t in run r, let $c_{(s,t,r)}$ be the number of detected `Final.py` updates and $v_{(s,t,r)}$ the number matched to successful verification records. If an update is matched, it indicates that the solution had been verified before submission; if it is not matched, no such prior verification was recorded. Averaging over $\mathcal{D}_s$, the runs with evaluable logs and $c_{(s,t,r)} > 0$, gives

$$\text{VR}_s = \frac{1}{|\mathcal{D}_s|} \sum_{(t,r) \in \mathcal{D}_s} \frac{v_{(s,t,r)}}{c_{(s,t,r)}}.$$

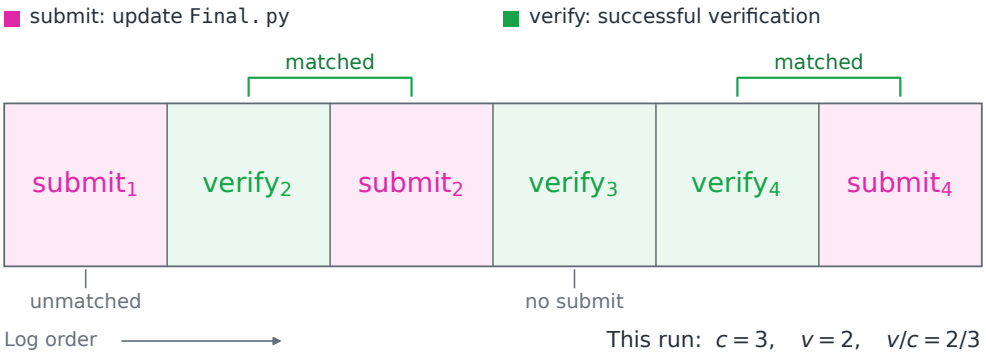


Figure 5: Verification-record matching in an example run. Magenta submit_i denotes an update to `Final.py`, and green verify_i denotes a successful verification record; the subscript identifies the solution. Solution 1 is submitted without a matching verification record, whereas submissions of solutions 2 and 4 are matched. Solution 3 is verified but never submitted. Thus, $c_{(s,t,r)} = 3$, $v_{(s,t,r)} = 2$, and this run contributes a fraction of $2/3$ to VR_s.

Because each term is a proportion, $\mathrm{VR}_s \in [0, 1]$. A higher VR therefore indicates a more rigorous submission habit: the agent tends to verify before submitting updates. A lower VR, by contrast, suggests that the agent often submits updates without prior verification.

Eligible solutions and analysis cohorts. The archived corpus contains 29,121 completed Golf-Unzip traces. We align traces with method signatures by solution identity and source hash, and both TR and MD metrics use the same eligible solution records. A run is eligible only when it contains at least two such records, so $n_{(s,t,r)} \geq 2$ for both scores. Missing or ineligible runs are excluded rather than assigned zero.

The *primary cohort* contains 92 tasks (15 easy, 46 medium, and 31 hard), each with at least one eligible run in every setting. Eligible run IDs may differ across settings. We average runs equally within each task and tasks equally within each setting. This cohort supplies the overall TR and MD scores, difficulty trends, balanced trick frequencies, and setting-preference heatmap.

The *matched cohort* contains 85 tasks with at least one common run containing two or more eligible solutions in every setting: 194 common task/run blocks in total. Harness comparisons first average paired Co–CC run differences within each task and then average tasks. Correlations and interaction tests use this same matched support. The *all-three-runs cohort* retains the 42 tasks for which all three run blocks are eligible in every setting and is used for sensitivity analysis. These restrictions concern completed, aligned candidate records; the cohort sizes are not counts of tasks solved by the agents. Appendix F.3 gives the statistical procedures and detailed estimates.

E.4 Full Results and Human-Reference Improvements

Improvements over the human reference. Verified agent solutions are shorter than the per-task minimum in the 48-team human reference on 10 of 96 tasks: seven medium and three hard. The Records column sums to 16 configuration–task improvements because multiple configurations can improve on the same task. Each entry uses the shortest verified program across runs for that configuration and task.

GRM-Rank comparisons also depend on the harness. Codex ranks ahead of Claude Code for Qwen (5 versus 15), GPT (9 versus 16), and DeepSeek (13 versus 15); Claude Code ranks ahead for GLM (13 versus 16). These ranks describe insertion positions on the calibrated human GRM scale, not the mean per-task human-relative ranks reported elsewhere. They summarize performance, whereas the interaction tests concern exploration among matched retained candidates.

Table 5: Overall evaluation. GRM-Rank is each setting's insertion position among 48 reference human teams on the calibrated GRM scale (lower is better). Trick Richness and Method Diversity (V) use the same 92 tasks with equal task and available-run weights. Records counts tasks with a verified solution strictly shorter than the per-task minimum in the 48-team reference; VR is the reported run-averaged fraction of final-file updates linked to successful verification records.

Setting	GRM-Rank	Trick Richness	Method Diversity	Records	VR
Co_Qwen	5	4.283	0.290	7	0.839
Co_GPT	9	4.471	0.359	4	0.884
Co_DS	13	3.832	0.309	3	0.908
Co_GLM	16	4.436	0.289	0	0.859
CC_Qwen	15	4.389	0.231	0	0.872
CC_GPT	16	4.288	0.261	0	0.943
CC_DS	15	3.987	0.325	1	0.897
CC_GLM	13	4.415	0.246	1	0.832

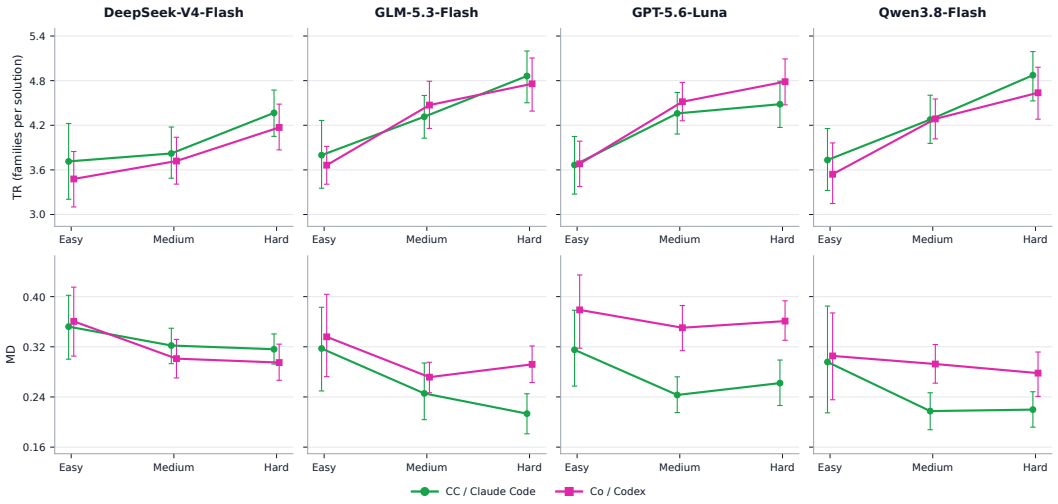


Figure 6: Difficulty trends in Trick Richness (top) and Method Diversity (bottom). Columns correspond to models; green circles denote Claude Code and magenta squares denote Codex. Points average eligible runs within each task, then tasks within each difficulty group. Bars show 95% task-bootstrap confidence intervals.

E.5 Supplementary Behavioral Analyses

Difficulty and method resolution. Figure 6 shows higher Trick Richness on harder tasks under both harnesses. Averaged over all eight settings, the primary hard–easy difference is +0.958 families (95% CI [0.518, 1.394]). Method Diversity V is lower on hard tasks in the primary cohort, with a difference of -0.0530 (95% CI $[-0.1010, -0.0053]$); its medium and hard means are similar. On the matched cohort, the hard–easy V difference is -0.0324 (95% CI $[-0.0954, 0.0321]$). The method-distance decrease is therefore less certain under common-run matching. Table 8 reports the primary estimates and supporting counts.

The distinct-method ratio K/n is lower on harder tasks, even though both the number of retained solutions and the number of distinct methods increase. A lower ratio therefore does not by itself indicate less exploration. Hard-task annotations also contain more `other` labels, which limits the resolution of the method rubrics. We interpret the observed V trend within this annotated method space, rather than as evidence of semantic convergence; the supporting counts and rubric diagnostics are reported in Appendix F.3.

Recurring trick families. Balanced frequency describes how often a family appears in an eligible solution while giving equal weight to settings, tasks, and available runs at their respective aggre-

gation levels. For family f, we first compute its solution prevalence within each run and average these prevalences within a task. Using $F^{(i)}_{(s,t,r)}$, $n_{(s,t,r)}$, and $R_{(s,t)}$ as defined for Trick Richness, let $p_{(s,t),f}$ denote this run-averaged prevalence for setting s on task t. On the task set T shared by all eight settings, define

$$
\begin{aligned}
p_{(s,t),f} &= \frac{1}{|R_{(s,t)}|} \sum_{r \in R_{(s,t)}} \frac{1}{n_{(s,t,r)}} \sum_{i=1}^{n_{(s,t,r)}} \mathbb{I}\{f \in F^{(i)}_{(s,t,r)}\}, \\
\bar{p}_{s,f} &= \frac{1}{|T|} \sum_{t \in T} p_{(s,t),f}, \\
\mathrm{BF}_f &= \frac{1}{8} \sum_{s=1}^{8} \bar{p}_{s,f}.
\end{aligned} \tag{19}
$$

Here $\bar{p}_{s,f}$ is the task-averaged prevalence for one setting, and BF_f averages that prevalence across settings. Each family is counted once per solution. Missing runs are excluded rather than imputed as zero. Consequently, runs with more solutions do not receive extra weight, and BF measures solution prevalence rather than the fraction of expansion steps. Table 6 reports the ten most frequent families and their cross-task coverage. List comprehensions and lambdas form a broadly shared core. The 10% coverage threshold distinguishes routine recurrence from a single isolated occurrence.

Table 6: Cross-task recurrence. BF is Eq. 19. *Any* counts tasks with at least one occurrence; $\geq 10\%$ counts tasks with $(1/8)\sum_s p_{(s,t),f} \geq 0.1$. Families are selected by BF, not by a setting contrast.

Trick family	BF (%)	Tasks: any	Tasks: $\geq 10\%$
List comprehension	81.0	92	92
Lambda	53.6	92	85
Sequence repetition	35.7	89	70
Short circuit	32.4	92	72
Unpacking	28.1	92	68
Boolean arithmetic	27.8	90	65
Starred unpacking	26.6	87	73
Generator expression	26.4	88	59
Walrus	24.1	88	76
Conditional expression	18.6	91	56

Configuration-specific trick preferences. Figure 7 shows each setting's percentage-point deviation from a family's eight-setting mean, separating relative preference from overall popularity. These preferences also appear in task/run-matched Co–CC comparisons. For example, Co increases generator-expression prevalence for GLM by 12.8 percentage points but decreases walrus prevalence for DS by 9.4 percentage points. Both contrasts survive Benjamini–Hochberg correction (Benjamini & Hochberg, 1995) across the 156 model–family comparisons. Table 9 reports the matched support, intervals, and corrected values.

Relationship between Trick Richness and Method Diversity. Trick Richness measures implementation-family richness within a solution, whereas Method Diversity measures distances among candidate methods. On the matched cohort defined above, the association is weak across setting averages and modestly negative after removing task and setting means from task-level scores. These patterns support complementary measurements without establishing statistical independence. For example, GLM's Method Diversity increases under Co, while its Trick Richness change is uncertain. Recovering more local tricks and exploring more distant methods need not move together (Table 9).

Model–harness interaction in exploration and performance. We compare Co and CC within the same task/run on 85 tasks and 194 matched blocks, each containing at least two eligible solutions in all eight settings (Figure 8a,b). Paired run differences are averaged within each task, then tasks receive equal weight. We test whether the four models have equal mean harness effects using 20,000

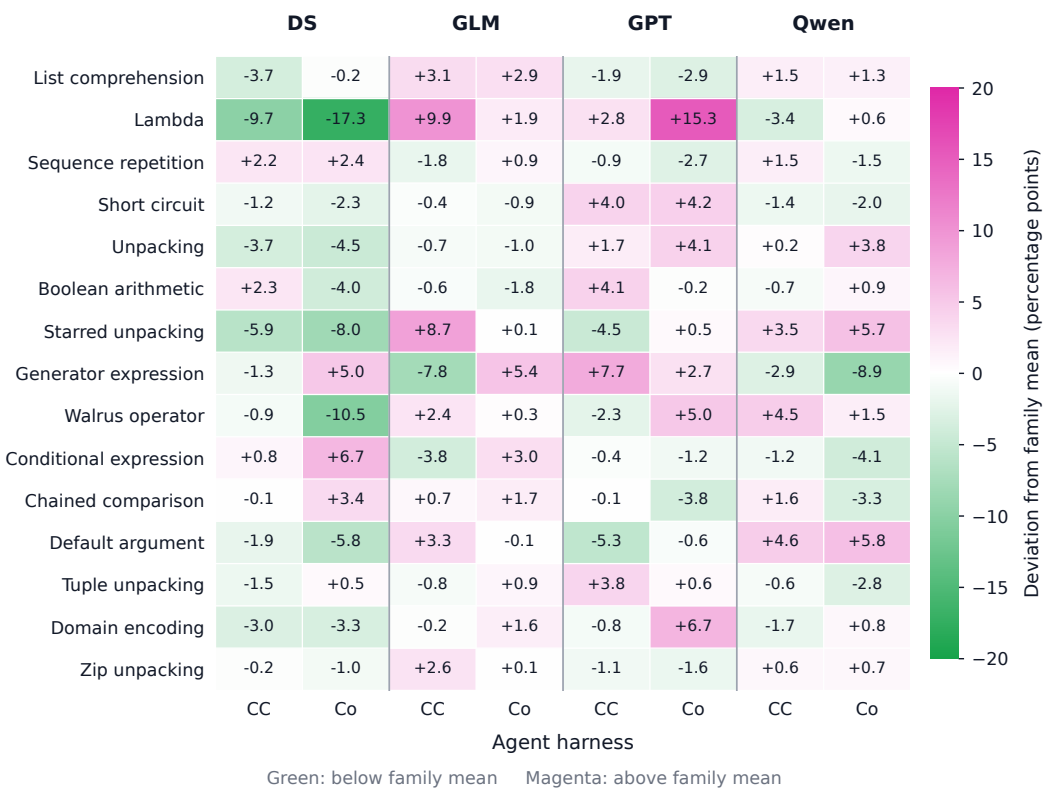


Figure 7: Setting preferences for the 15 most frequent families. Cells show percentage-point deviations from the family's balanced frequency. Magenta denotes above-average prevalence and green below-average prevalence; here color encodes the deviation, rather than the harness.

task-level wild-bootstrap replicates. Table 9 reports all eight model-specific effects and their 95% CIs, obtained by resampling tasks while retaining their matched runs and settings.

For Trick Richness, Co increases GPT's score but decreases DS's. Their harness effects differ by 0.455 families per solution (95% CI [0.210, 0.698]; $q = 0.0024$, Benjamini–Hochberg correction (Benjamini & Hochberg, 1995) across six model pairs within this metric). For Method Diversity V, Co's effect is largest for GPT, positive for GLM and Qwen, and inconclusive for DS. Omnibus tests support interaction for Trick Richness ($p = 0.0012$) and V ($p < 0.0001$). Requiring all three eligible runs in every setting leaves 42 tasks: the V interaction persists ($p = 0.00035$), while the Trick Richness evidence weakens ($p = 0.055$).

GRM-Rank provides a separate descriptive comparison (Figure 8c). It uses each setting's shortest verified program per task across runs on the 96-task benchmark, omits CC–GPT's single missing response from its likelihood, and places settings among 48 human teams on the fixed GRM scale; lower ranks are better. Co ranks ahead for Qwen (5 versus 15), GPT (9 versus 16), and DS (13 versus 15); CC ranks ahead for GLM (13 versus 16). The TR and MD tests concern matched retained candidates, whereas GRM-Rank describes calibrated performance from best programs. These results support model-specific and metric-specific harness conclusions.

E.6 Radar Visualization

The radar chart in Figure 9 summarizes GRM-Rank, Trick Richness, distinct-method count K, Method Diversity V, and VR. It is a descriptive profile rather than a composite score. The observed method count K depends on the number of retained candidates.

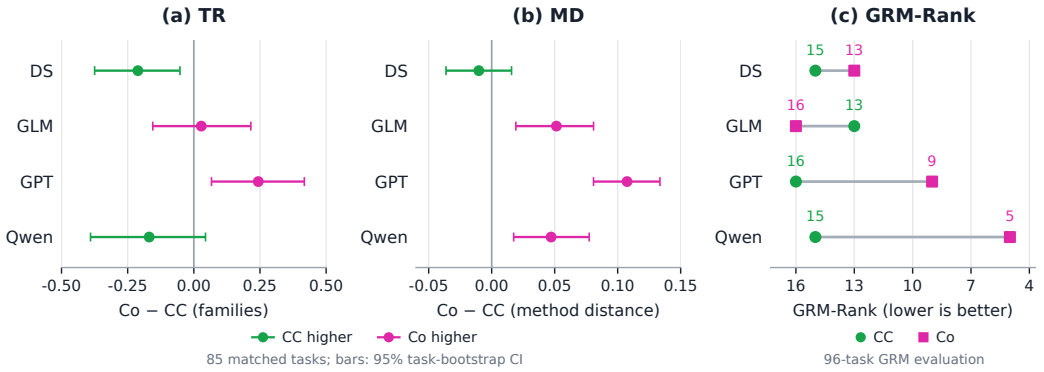


Figure 8: Model-specific harness effects. (a,b) Co–CC differences on common task/run blocks with at least two eligible solutions in every setting. Differences are averaged within tasks and then across tasks; bars show 95% task-bootstrap confidence intervals. (c) GRM-Rank from each setting's shortest verified programs across runs on the 96-task benchmark. Lower ranks are better.

For metric j, let $x_{s,j}$ denote its value for configuration s, and let μ_j and σ_j be its mean and population standard deviation across the eight configurations. We standardize each value as

$$z_{s,j} = (x_{s,j} - \mu_j)/\sigma_j$$

and plot

$$\left[1 + \exp(-d_j z_{s,j})\right]^{-1},$$

where $d_j = -1$ for GRM-Rank and $d_j = 1$ otherwise. Lower GRM-Rank and higher values of the other measures point outward; the 0.5 ring corresponds to each metric's mean across configurations. Outward position does not imply that every metric is a performance objective.

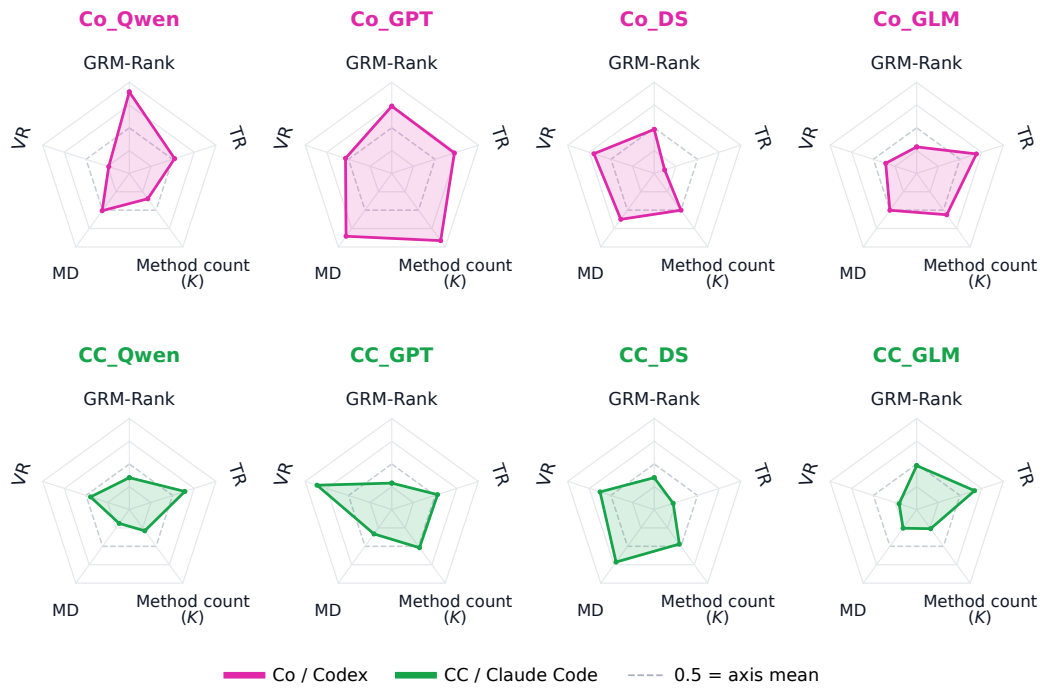


Figure 9: Behavioral profiles on five standardized measures. The 0.5 ring marks the mean across configurations; the profiles do not define a composite score.

Measurement limitations. The analysis describes retained candidates through archived expansion and annotation outputs. Recovered labels summarize an expansion path and need not reconstruct the original generation history. Equal task and run weights prevent candidate-rich runs from dominating the aggregate, but do not remove selection effects from candidate retention or trace completion. Variation in expansion protocols and task-specific rubrics also limits comparisons. In particular, Trick Richness measures recovered richness per solution, Method Diversity measures diversity within the annotated space, and VR measures verification linkage for final-file updates.

F GOLF-UNZIP EXPANSION PROTOCOL

Golf-Unzip constructs a sequence of verified expansions from a compressed source solution. Here, `judge` denotes the expansion judge, distinct from the agent setting that originally generated the solution. The judge is instructed to expose one compression mechanism per proposal. The runner accepts only candidates that pass the configured task-verification cases.

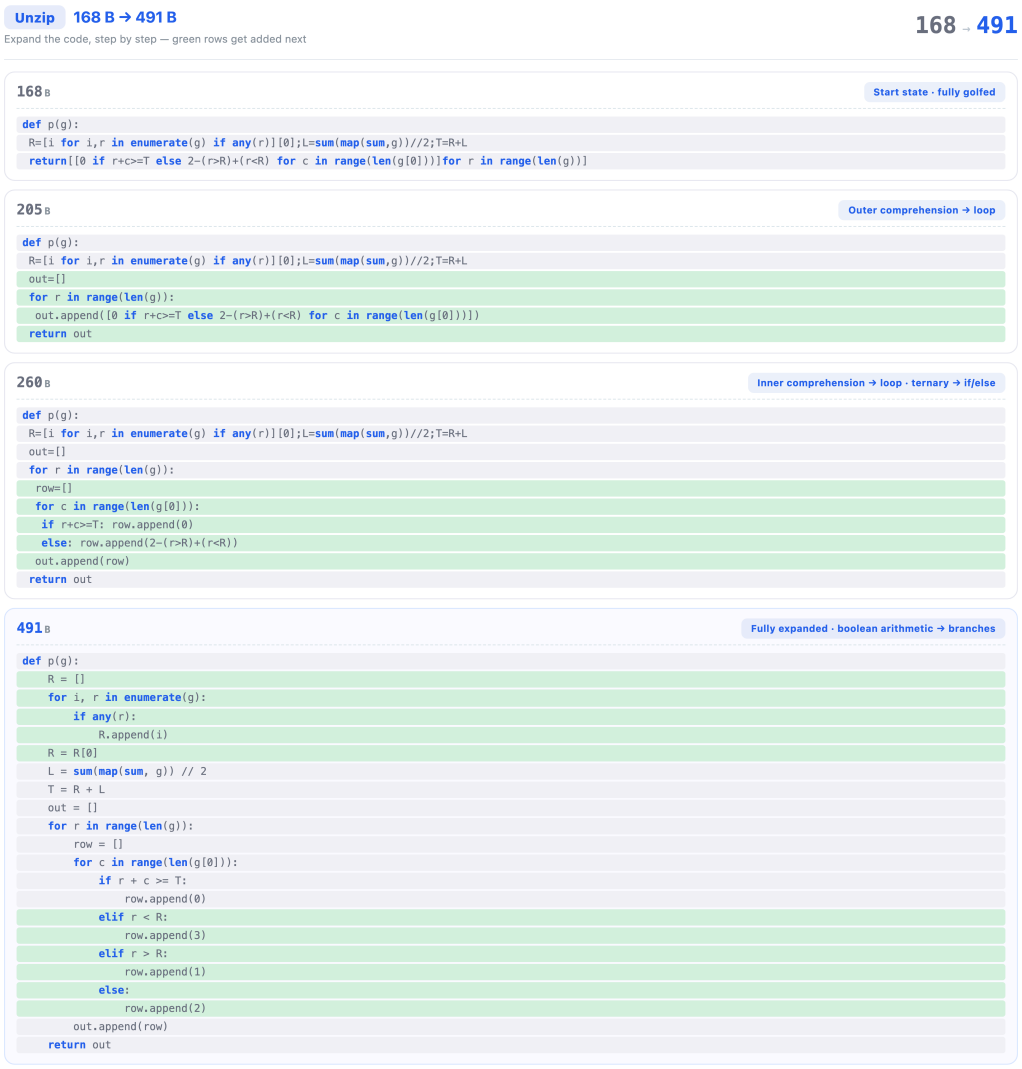


Figure 10: An example Golf-Unzip sequence, showing accepted expansions and their labels.

Verification and accepted states. VERIFY returns PASS only when the candidate passes all configured task cases, with the expected case count checked when supplied. A source that fails these

Algorithm 1 Golf-Unzip core runner: run($record$)

Default parameters: $max_steps = 64$, $max_retries = 2$.
$code \leftarrow$ READ_SOURCE_CODE($record$)
$initial \leftarrow$ VERIFY($code$)
if $initial =$ ERROR **then return** ERROR
if $initial \neq$ PASS **then return** SKIP
$steps \leftarrow 0, \quad retries \leftarrow 0, \quad feedback \leftarrow \emptyset$
while true **do**
  if $steps \geq max_steps$ **then return** BLUR
  $raw \leftarrow judge$.ask(PROMPT($code, steps + 1, feedback$))
  if the API request fails **then return** ERROR
  $proposal \leftarrow$ PARSE_JSON(raw)
  if $proposal.status =$ BLUR **then return** BLUR
  if $proposal.status =$ DONE **then**
    if RESIDUAL($code$) $= \emptyset$ **then return** DONE
  else if $proposal \neq \perp$ **then**
    $candidate \leftarrow$ MAP_REPLACEMENT($code, proposal$)
    if $candidate \neq \perp \wedge candidate \neq code$ **then**
      $candidate \leftarrow$ ENSURE_IMPORTS($candidate$)
      if VERIFY($candidate$) = PASS **then**
        ACCEPT($candidate$); $code \leftarrow candidate$
        $steps \leftarrow steps + 1, \quad retries \leftarrow 0$
        $feedback \leftarrow$ passing verification; **continue**
  $retries \leftarrow retries + 1, \quad feedback \leftarrow$ FAILURE_DIAGNOSTICS
  if $retries > max_retries$ **then return** BLUR

Here, $retries$ counts failed proposals, including the initial failure. The pseudocode describes uninterrupted execution of the short-prompt (old) runner; resume behavior is specified below.

tests is skipped; a verifier infrastructure or configuration failure at preflight is reported as ERROR. Candidate verification failures leave the accepted program unchanged and produce rejection feedback. Passing the finite verification suite establishes agreement on those cases, rather than equivalence on every possible input. In the pseudocode, ACCEPT records the verified replacement and its step metadata; rejected attempts are not expansion steps. This expansion-runner operation is separate from the agent's original updates to `Final.py` evaluated by VR.

Proposal mapping. PARSE_JSON accepts one JSON object and returns $\perp$ for a malformed reply; the status of $\perp$ is undefined and matches neither DONE nor BLUR. MAP_REPLACEMENT first accepts a replacement that parses as a complete program defining `p`. Otherwise, it inserts the replacement suite at the requested statement target, normalizing its indentation; the suite may contain multiple statements. An empty replacement yields $\perp$; in the statement branch, an unknown target also yields $\perp$. Statement targets are rebuilt from the current program for each proposal. The runner rejects a textually unchanged candidate and uses configured import hints to supply missing imports before verification. These checks do not prove that exactly one semantic mechanism changed, and no additional cycle-detection rule is assumed here.

Completion and stopping. RESIDUAL($code$) scans for supported compression constructs, including comprehensions, generators, lambdas, walrus assignments, starred unpacking, conditional expressions, chained comparisons, syntactically evident sequence repetition, and default arguments. This scan is a lower bound: a nonempty result blocks DONE, whereas an empty result does not establish semantic completeness. A model DONE declaration is accepted only when the scan is empty; otherwise it is rejected and consumes the same proposal-failure budget as parsing failures, unusable or unchanged replacements, and failed candidate verification. Model-declared BLUR terminates immediately. API failures terminate as ERROR; transport-level retries are separate from proposal retries.

The short-prompt (old) runner defaults to `max_steps = 64` and `max_retries = 2`. Only verified, accepted expansions count toward the step limit. The step-limit check occurs before the next model request: after the 64th accepted expansion, the trace terminates as BLUR without an additional opportunity for the model to declare DONE, irrespective of the residual scan.

The proposal-failure counter includes the initial failed attempt. With `max_retries = 2`, the runner allows an initial attempt and up to two retries at the same accepted state, terminating as BLUR after three consecutive failures. An accepted expansion resets this counter to zero. In this runner, resuming a RUNNING trace also resets the counter, so the three-failure limit applies within an uninterrupted execution segment. The stopping reason is recorded, and the last accepted program remains available for an incomplete trace. Only DONE traces enter the TR and MD analysis; BLUR and ERROR traces are excluded even when they retain accepted expansions.

F.1 Expansion Prompt

The judge model is driven by the following prompt. At each step it must undo exactly one semantic trick and emit a short free-text label naming that mechanism; the external runner, not the model, verifies every proposed state against the complete task verifier.

Expansion Prompt and Vocabulary (RESTORE_TRACE)

```
PROMPT = """\
Restore an already verified Python code-golf solution.
Return exactly one JSON object. Make one semantic expansion per step.
    Choose
trick_code from the fixed vocabulary; use OTHER only when no code fits.
The runner supplies statement targets. Return a replacement for exactly
    one
target statement; the replacement may contain several statements when
    that is
the direct expansion of that one target.
Return a standalone Python suite: first line and every top-level
    statement at
column zero, indent only nested bodies. Choose the smallest adequate
    target.
Preserve unrelated operations, evaluation order, mutable state and
    aliasing.
Do not rename unrelated variables or invent a different algorithm. A
    named
function target may be necessary to lift an expression inside a lambda.
Expand comprehensions, generators, lambdas, walrus, starred unpacking,
    chained
comparisons, ternaries, hidden default state and sequence repetition.
    Also
expose short-circuit value selection and boolean arithmetic. Keep
    ordinary
arithmetic, calls, indexing and slicing when they no longer hide control
    flow
or construction. Do not expand normal builtins merely to increase step
    count.
Known residual syntax is a lower bound: nonempty means no DONE; empty
    requires
a semantic review. A new mechanism may use OTHER and does not require
    BLUR.

Step response:
{"status":"PROPOSE_STEP", "target_id":"...", "replacement":"...",
 "trick_code":"COMP_LOOP", "target":"list_comprehension",
 "evidence":"short reason"}

Completion response:
{"status":"DONE", "unresolved_items":[], "reason":"..."}

If one reliable step cannot be identified:
{"status":"BLUR", "reason":"..."}
"""

VOCABULARY = """\
COMP_LOOP: comprehension/generator to explicit loop
```

```
LAMBDA_DEF: lambda to named function
BINDING: walrus/default state to assignment
CONTROL_EXPR: conditional/short-circuit expression to control flow
REPEAT_BUILD: sequence repetition to explicit construction
UNPACK_LAYOUT: unpacking/transpose to explicit layout
BOOL_ARITH: boolean or arithmetic shortcut to explicit logic
DOMAIN_ENCODING: regex/string encoding to explicit algorithm
OTHER: verified mechanism outside this vocabulary"""
```

F.2 Trick Lexicon and Normalization

Each accepted expansion step stores three annotation fields with different roles:

- `trick_code`: a coarse code requested from the prompt's nine-code vocabulary;
- `label`: a free-text description of the compression mechanism;
- `label_family`: the archived family identifier used by Trick Richness.

The exporter provides string normalization, explicit aliases, and ordered keyword rules. The reported analysis reads the family assignments already stored in each trace; it does not recompute them from the raw labels. Table 7 lists the 39 scored family identifiers and representative operational descriptions. The additional `noise` category is excluded from the score.

Fix a setting–task–run combination (s, t, r) and consider its eligible solution i. Let S_i be the accepted expansion-step indices in this solution's completed trace, and let $z_{i,k}$ be the archived `label_family` at step k. Within this fixed combination, the family set used in the main-text Trick Richness definition is

$$F^{(i)}_{(s,t,r)} = \{z_{i,k} : k \in S_i\} \setminus \{\texttt{noise}\}.$$

The solution's family count is $|F^{(i)}_{(s,t,r)}|$; averaging these counts over solutions gives $C_{(s,t,r)}$. Repeated steps from one family contribute only once. In pseudocode:

```
def solution_families(trace):
    assert trace["status"] == "DONE"
    families = {step["label_family"] for step in trace["steps"]}
    families.discard("noise")
    return families

family_count = len(solution_families(trace))
```

These descriptions summarize the archived codebook, rather than defining a new string-matching function or mutually exclusive Python syntax classes. Related mechanisms may overlap conceptually; each step retains its archived family assignment. Thus, the analysis does not infer a family anew from a raw label's spelling, and `noise` is treated as an excluded annotation category rather than proof that a step contains no meaningful mechanism.

Table 7: The 39 scored family identifiers and representative operational descriptions. Identifiers are preserved from the archived expansion steps; the additional `noise` category is excluded from Trick Richness.

Family identifier	Representative operational description
`list_comprehension`	List comprehensions, including comprehension-based flattening, expanded into loops
`set_comprehension`	Set comprehensions expanded into explicit construction
`dict_comprehension`	Dictionary comprehensions expanded into explicit construction
`generator_expression`	Generator expressions and associated yield-based mechanisms made explicit
`lambda`	Lambda expressions rewritten as named functions
`sequence_repetition`	Sequence repetition, such as `[x]*n`, expanded into explicit construction
`walrus`	Assignment expressions using `:=`
`default_argument`	Default arguments used for compact binding or persistent state
`chained_comparison`	Chained comparisons, such as `a<b<c`
`short_circuit`	`and`/`or` used for control flow or value selection
`conditional_expression`	Conditional selection, boolean indexing, membership tests, and default lookups
`boolean_multiplication`	Boolean values used as numeric multipliers
`boolean_xor`	Exclusive-or operations on boolean values
`truthiness_to_numeric`	Implicit truth tests or truth-value coercions made explicit
`bitwise_arithmetic`	Integer bitwise operations used as arithmetic shortcuts
`boolean_arithmetic`	Arithmetic involving boolean values or boolean-derived quantities
`slice_step`	Stepped slices and related compact slicing operations
`flattening`	Nested-sequence flattening through concatenation or iteration
`domain_encoding`	String, byte, regex, `exec`, or `eval` encodings of a computation
`map`	`map()` calls expanded into explicit iteration
`zip_unpacking`	`zip()` combined with unpacking or transposition
`enumerate_unpacking`	`enumerate()` combined with unpacking
`tuple_unpacking`	Tuple unpacking in assignments or loops, including swaps
`starred_unpacking`	Starred unpacking in calls or container construction
`unpacking`	General unpacking, transposition, and associated flat-index layout rewrites
`singleton_loop`	Single-element loop bindings, such as `for x in [expr]`
`chained_assignment`	Chained assignments, such as `a=b=c`
`augmented_assignment`	Augmented assignments, such as `+=` or `*=`
`predicate_builtin`	Predicate builtins such as `filter()` and `any()`
`sequence_building`	Sequence construction and concatenation
`for_else`	`for...else` control flow
`reduction_builtin`	Reductions, including `min`/`max` with keys, and helpers such as `divmod`
`itertools`	Iterator utilities such as `groupby`, `accumulate`, or `product`
`set_operation`	Set intersection, union, and subset operations
`dict_dispatch`	Dictionary lookup tables used for dispatch or value selection
`index_search`	Index searches and related indexing shortcuts, including negative indices
`sentinel_search`	Sentinel values used to terminate a search or mark a condition
`recursion`	Recursive calls used as a compact control structure
`dynamic_loop`	Iteration over a mutating or dynamically growing container

F.3 Detailed TR and MD estimates and statistical procedures

The main difficulty and preference analyses use the same 92-task cohort (15 easy, 46 medium, and 31 hard). Each setting has at least one eligible run per task, but run IDs may differ across settings. We average eligible runs within a task and then average tasks, assigning equal weights at each level. Confidence intervals are percentile 95% task-bootstrap intervals. Difficulty contrasts resample the disjoint task groups separately; intervals describe task heterogeneity conditional on the archived annotations.

The mean Trick Richness for medium tasks is 4.221 and the mean Method Diversity V is 0.2805; all three difficulty groups are shown in Figure 6. The task-balanced fraction of method-axis labels assigned to `other` is 3.4% for easy tasks and 19.9% for hard tasks in the full corpus. This rubric diagnostic uses the full corpus rather than the primary 92-task comparison cohort and has no inferential test added here. The reported K/n averages within-run ratios and is not the ratio of the

Table 8: Difficulty estimates underlying the main-text discussion. The hard–easy comparison uses 31 hard and 15 easy tasks from the primary 92-task cohort. All eight settings receive equal weight. n is the retained-solution count per run; K is the number of distinct complete method signatures. Intervals are existing 95% task-bootstrap intervals.

Measure	Easy	Hard	Hard–easy	95% CI
Trick Richness	3.660	4.617	+0.958	[0.518, 1.394]
Method Diversity V	0.3326	0.2797	-0.0530	[-0.1010, -0.0053]
Distinct-method fraction K/n	0.829	0.563	-0.265	[-0.330, -0.199]
Retained solutions n	5.94	15.54	+9.60	[7.53, 11.62]
Distinct methods K	3.86	7.17	+3.312	[2.339, 4.296]

displayed mean K to mean n. The lower K/n on hard tasks accompanies more candidates and more distinct methods; it should not be read as an isolated measure of exploration breadth.

Table 9: Detailed estimates for the matched analyses. Panels A–D use 85 tasks and 194 common task/run blocks, each containing all eight settings with at least two eligible solutions. Effects are Co minus CC unless specified otherwise; pp denotes percentage points. Panel E separates the primary 85-task interaction tests from the 42-task subset with all three run blocks eligible in every setting. CIs are 95% task-bootstrap intervals. A dash indicates that no value is reported in that column.

Comparison	Estimate	95% CI	Test value
A. Family prevalence (BH across 156 model–family tests)			
GLM: generator expression (pp)	+12.8	[7.2, 18.6]	$q = 0.0039$
GPT: lambda (pp)	+11.1	[5.3, 17.0]	$q = 0.0201$
DS: walrus (pp)	-9.4	[-13.8, -4.9]	$q = 0.0039$
B. Model-specific harness effects			
Trick Richness: Qwen	-0.169	[-0.390, 0.044]	—
Trick Richness: GPT	+0.244	[0.067, 0.418]	—
Trick Richness: DS	-0.211	[-0.375, -0.052]	—
Trick Richness: GLM	+0.028	[-0.155, 0.216]	—
Method Diversity V: Qwen	+0.047	[0.017, 0.077]	—
Method Diversity V: GPT	+0.107	[0.081, 0.134]	—
Method Diversity V: DS	-0.010	[-0.036, 0.016]	—
Method Diversity V: GLM	+0.051	[0.019, 0.081]	—
C. Difference between harness effects (BH across six model pairs)			
Trick Richness: GPT minus DS	+0.455	[0.210, 0.698]	$q = 0.0024$
D. Pearson association between Trick Richness and Method Diversity V			
Eight setting means	-0.179	[-0.400, 0.078]	—
Task-and-setting residuals	-0.106	[-0.194, -0.014]	—
E. Omnibus interaction (task-level wild-bootstrap p values)			
Trick Richness, 85 tasks	—	—	$p = 0.0012$
Method Diversity V, 85 tasks	—	—	$p < 0.0001$
Trick Richness, 42 tasks	—	—	$p = 0.055$
Method Diversity V, 42 tasks	—	—	$p = 0.00035$

The matched analyses first average Co–CC run differences within each task. The omnibus null hypothesis is equality of mean harness effects across the four models, calibrated with 20,000 task-level wild-bootstrap replicates. Family and model-pair contrasts use their existing sign-flip tests with Benjamini–Hochberg correction (Benjamini & Hochberg, 1995) applied separately to the 156-family and six-model-pair test families. The q values in Panels A and C are adjusted values; the omnibus p values in Panel E are not presented as BH-adjusted values. Correlation intervals also resample tasks; the eight-setting correlation is descriptive and does not establish independence of the metrics. The 42-task restriction changes the analysis support and provides weaker evidence for the Trick Richness interaction.

Supplementary method summaries. The distinction between sampling records with and without replacement matters at finite n: multiplying the with-replacement Rao quadratic expression by $n/(n-1)$ gives the pair mean used here.

For a fixed setting–task–run combination, we additionally report K, the number of distinct complete signatures, and K/n, its ratio to the number of eligible solution records. We also report the expected number of distinct signatures in a uniformly sampled set of five records drawn without replacement. This expectation is computed by exact rarefaction only for runs with $n \geq 5$. These supplementary summaries do not replace the primary pairwise-distance score. All method comparisons remain conditional on the task-specific rubrics, including the retained `other` category.

G Reproducing the skill-transfer experiment

We record the implementation details needed to repeat Section 6, e.g., the library at every version, the prompts verbatim, the harness, and the per-task solutions behind every number. We present the *logic* and *principle* of the setup.

Task split. The benchmark's tasks are referred to by their three-digit index. Library version n was built from skill sets $1..n$ and nothing else; The skill set pool and the ten hold-out tasks are disjoint (see Table 10). The hold-out set is fixed before library construction.

Table 10: The task split of skill sets and hold-out set.

skill set 1	002, 021, 025, 026, 028, 031, 049, 050, 090, 091
skill set 2	003, 004, 032, 034, 035, 037, 038, 039, 051, 074
skill set 3	005, 006, 019, 042, 043, 046, 053, 064, 094, 095
skill set 4	008, 010, 018, 054, 055, 056, 061, 063, 066, 079
skill set 5	014, 015, 022, 023, 041, 075, 076, 081, 086, 087
hold-out set	**001, 011, 027, 048, 062, 067, 072, 078, 089, 096**

What the protocol must protect, and how. Four things have to be protected for a valid knowledge-transfer measurement, and each maps to one mechanism in the harness.

Leakage of reference solutions: the staging step copies exactly five files per task (`Prompt.md`, `gen.py`, `common.py`, `verify.py`, `data.json`), so the best agent and human solutions that sit beside them in the benchmark can never enter a sandbox.

Cross-session contamination: every session runs in a throwaway directory with its own temporary space, and an audit hashes the task files before and after, compares the two arms' finals for byte-identity, and scans logs for out-of-scope paths.

Self-reported scores: the harness extracts the declared solution and re-runs the official verifier on it in a fresh copy of the task with `data.json` made read-only (the verifier refuses to run otherwise); the byte count is the verifier's own.

A second variable hiding in the prompt — the two arms' prompts are generated from one template and differ in a single block ("the library is `./skill_library.json`; read it whole; here is how its three levels work" versus "no reference material is provided"); the rendered prompts for every session type are in the repository so the diff can be checked.

Why lanes are assigned by the harness. The DeepSeek v4 Flash executes a chosen representation family competently but does not generate cross-family alternatives on its own, and the library's recommendation can anchor it in the wrong one. The harness therefore offers a configuration in which the family is an *input* to a session: the treated arm runs one session per lane (the library's own L3 families: candidate enumeration with scored selection; recursion and rotation skeletons; direct construction and separable folds), each carrying the lane's slotted skeleton, the entries to read first, and a commit-unless-provably-infeasible clause; the control arm runs the same number of sessions with content-free diversity instructions (most natural formulation; a different program

Table 11: The three lanes assigned to the treated arm. Every task receives every lane, one session each; the harness keeps the shortest verified result and records which lane produced it.

lane	fits when	skeleton (slots in capitals)
A enumeration	the answer is one of a small enumerable set (a window, a crop, a period, a parameter combination) chosen by a score or an exact check	`p=lambda g:max((SCORE(c),c)for n in range(N) if(c:=BUILD(g,n)))[-1]`
B rotation / recursion	the rule is applied in several directions, or the generator flips or transposes at random, or the output is N passes of one transformation	`p=lambda g,n=3:g*-n or p([[r.pop()or F(r)for r in g]for r in g],n-1)` and the slice-recursion and walrus-pass variants
C dimensional recursion / separable fold	the row operation equals the column operation, or each cell is a closed form of its line and one located reference cell	`p=lambda g:g*0!=0and[*map(p,g)]or LEAF(g)` and the neighbour-channel and depth-2 variants

shape; a different data representation), so compute is matched and no family knowledge crosses arms.

All candidates are verified and the shortest passing one per task is kept, with its lane recorded: the winning-lane tally describes which assigned families produce the shortest verified candidates. The single-session configuration and the lane configuration are both one command each; the repository's `README` gives them, together with the settings (model strings, endpoint, timeouts, concurrency) under which the reported numbers were produced.

What a lane is. A lane is a *program shape*, not a solution: a fixed skeleton with named slots, a one-line condition for when the shape fits, a short note on how to fill the slots, and the four or five library entries to read first (Table 11). The three lanes are coarse groupings of the library's own representation ladder (L3-03), which lists eighteen shapes with their measured fixed costs; A collects the enumeration shapes, B the rotation and recursion pipelines, C the per-axis and dimensional-recursion shapes. The task-specific part — the predicate, the leaf, the pattern — is what the session must still derive from the generator, and it is the part that decides the byte count.

How the lanes were produced. Nothing in the lane definitions is generated by a model or chosen per task; the file is fixed and identical for all ten tasks. Its provenance has two steps. The partition into A, B and C, the fitting conditions and the slot structure come from an earlier forced-diversity experiment of ours on a *different* task set, written before the present hold set was ever run; the families are generic shapes of Python grid programs and would read the same for any benchmark of this kind. For this study the file was re-pointed at library version 6: the skeletons were replaced by v6's, the quoted fixed costs by v6's measurements (obtained by executing each skeleton with its slots removed, a task-independent procedure), and the entry references by v6's identifiers. The text of the notes was aligned with v6's rule for the order in which to try per-cell rules inside a rotation pass.

Can the lanes leak hold-set knowledge? Three channels have to be considered, and they differ in severity. *Solution content*: none. The lanes contain no task-specific predicate, leaf, constant or pattern, and a provenance audit of the library builders confirmed that the one solution idiom that had entered the library from a hold trajectory was removed before v6; idioms seen in the hold trajectories were deliberately kept out of the v6 lane table from which the lanes are drawn. *Selection*: structurally impossible. A lane assignment could encode "task x wants family y" only if the assignment varied by task; here every task receives every lane, so the file cannot tell a session which family its task wants — that is precisely the decision the design removes from the model, and the winning-lane tally is measured after the fact, not written in. *Process hints*: present, and declared. Two sentences in the notes — lane C's "audit this lane even when the task looks like kron, crop or index arithmetic" and lane B's rule order (stateless predicate, then state carrier, then regex) — were written for v6 after we had read the hold trajectories of earlier library versions and seen sessions dismiss those

Table 12: Holdout-set results, bytes per verified solution (lower is better); all ten tasks solved in every setting. Both workers receive the same library. Worker A uses one session per task and condition. Worker B reports the adapted-harness comparison in Section 6.3, with three sessions per task and condition, and reports best of three verified solutions. References: the best agent solution previously recorded for each task (best of 24 runs over 8 agent configurations) and the best known human solution.

	Worker B (adapted harness)		**Worker A**		**References**	
task	no lib	with lib	no lib	with lib	prior agent	human
T001	104	99	94	101	93	93
T011	48	**46**	58	50	46	46
T067	114	94	99	95	89	83
T027	115	**81**	119	**81**	106	82
T072	107	89	100	78	97	74
T048	101	102	101	94	94	86
T078	184	151	169	120	146	110
T089	122	145	108	111	121	81
T062	159	146	116	99	89	89
T096	63	59	59	59	59	43
total	**1117**	**1012**	**1023**	**888**	940	**787**
vs. human	+42%	+29%	+30%	+13%	+19%	—

families on sight. They name no task and no solution, and they are the same category of item that the library's provenance audit flags in L3-03 itself, but a reader should know that the lane *notes* are not clean of hold-set process influence even though the lane *skeletons* and *partition* are. The clean control is cheap and we recommend it: run the lane configuration with the notes stripped (skeleton, fitting condition and entry references only), or with the original lane text of the earlier experiment mapped to v6 identifiers, and check that the winning-lane tally and the totals survive. What cannot leak through the lanes under any of the three channels is the thing the experiment measures: whether a session that has been told the family can execute it.

What is retained. Per session: the prompt, the full log with its token and cost line, the produced `Final.py`. Per run: the per-candidate verification table, the scoreboard with the per-session spread, the winner per cell with lane attribution. Every number in Section 6 can be recomputed from the retained solutions without calling a model.

G.1 Results

Effects of the evaluated interventions. Table 12 reports the library comparison for Worker A and the combined library-and-harness comparison for Worker B. For Worker A, total program length decreases from 1023 to 888 bytes with the library (13.2%), with seven wins, one tie, and two losses. For Worker B, skill-guided, family-assigned parallel search yields 1012 bytes, compared with 1117 bytes for the library-free generic parallel control (9.4%), with eight wins and two losses. Both achieve 81 bytes on T027, beating the human reference by one byte, although their aggregate lengths remain 13% and 29% above the human total, respectively.