SKILLSCRIPTBENCH: BENCHMARKING SELF-EVOLUTION OF EXECUTABLE AGENT SKILL PACKAGES BEYOND MARKDOWN

Anonymous authors
Paper under double-blind review

ABSTRACT

Executable Agent Skills combine natural-language instructions and scripts into reusable packages for LLM agents, and revising them requires fixing errors without breaking correct behavior. Existing benchmarks do not systematically distinguish documentation repair, script repair, and preservation when evaluating skill self-evolution. We introduce SkillScriptBench, a 350-task benchmark designed to evaluate these capabilities separately. From a survey of over 35,000 GitHub-hosted Skill roots, we select 100 packages and construct 150 repair tasks. Each task pairs a package containing injected script faults with a maintenance request and executable checks of the required behavior. A complementary controlled track contains 200 tasks from 50 packages, each evaluated under the same maintenance request in four states: clean, documentation faults, script faults, and faults in both. Across four LLMs, methods that edit both documentation and scripts can repair script faults but do not consistently outperform Markdown-only revision on documentation repair or preservation. We therefore introduce AST-Guided Skill Revision, which uses abstract syntax trees and calling relationships to link maintenance requirements to relevant code locations. It restricts script edits to these locations and updates the documentation to match the revised scripts. Averaged across models, this revision stage yields absolute gains in repair success of 21.9% for Raw Package and 27.7% for CoEvoSkills on faulty packages. Absolute gains in the proportion of tasks solved in all three runs reach 20.8% and 31.5%, respectively, indicating more consistent repair success across repeated runs.

1 INTRODUCTION

LLM agents increasingly perform multi-step tasks through sustained interaction with tools, files, and execution environments (Yao et al., 2023; Yang et al., 2024). Reliable execution depends on procedural knowledge for coordinating operations, recovering from failures, and validating outputs (Wang et al., 2023; Shinn et al., 2023). Agent Skills make this knowledge reusable by organizing instructions in a `SKILL.md` file, optionally accompanied by executable scripts and supporting resources (Agent Skills, 2026; Li et al., 2026). These packages provide workflows that agents can reuse across tasks, while skill self-evolution uses experience and feedback to revise them as requirements change or failures emerge (Zhong et al., 2026; Zhang et al., 2026b; Liu et al., 2026a;b). For executable skills, this process involves both the documented workflow and its implementation, making consistency between documentation and scripts an important part of reliable evolution.

Existing research spans three settings: evaluating the downstream utility of fixed skills, generating reusable skills from task experience or source corpora, and refining existing skills through feedback. Fixed-skill benchmarks assess packages without revising them, while representative skill-generation and continual-learning benchmarks often use a `SKILL.md` document as the learned artifact (Li et al., 2026; Han et al., 2026; Zhong et al., 2026; Zhou et al., 2026). Among self-evolution methods, SkillOpt and SkillAudit primarily revise instruction documents, whereas CoEvoSkills and SkillEvolver support executable multi-file packages (Yang et al., 2026; Gao et al., 2026; Zhang et al., 2026b;a).

Despite this progress, the self-evolution of existing executable skill packages remains underexplored in current benchmarks, particularly the revision of scripts and their consistency with documentation.

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

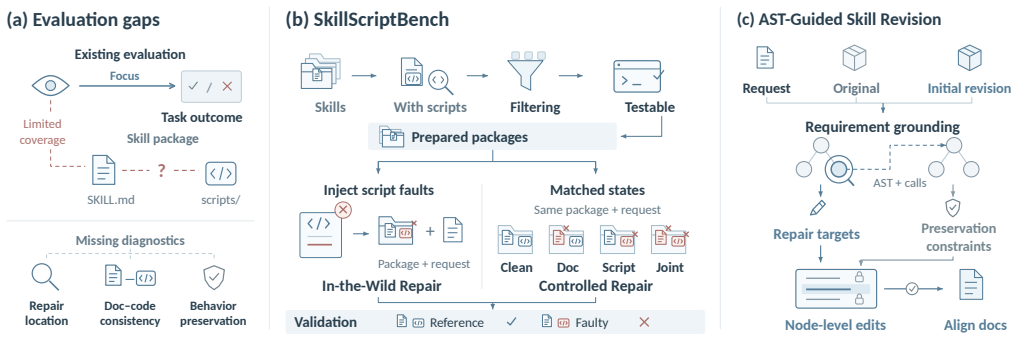

Figure 1: Overview of SKILLSCRIPTBENCH and AST-Guided Skill Revision: evaluation gaps, benchmark construction, and coordinated documentation–script revision.

These components can fail independently: a documented command may pass an unsupported option to an otherwise correct script, while a script may mishandle an option that is correctly documented. Task success and broader skill-quality assessments do not, by themselves, distinguish documentation repair from script repair, and aggregate results can obscure regressions in packages that were already correct (Li et al., 2026; Zhong et al., 2026; Yang et al., 2026; Zhang et al., 2026b). Addressing these gaps requires a benchmark that evaluates revisions for fault repair, documentation–script consistency, and preservation of correct behavior across diverse executable packages.

To address these gaps, we introduce SKILLSCRIPTBENCH, a 350-task benchmark that treats existing executable Agent Skill packages as the unit of self-evolution and evaluation. Each task pairs a package with a maintenance request. For the In-the-Wild Repair track, we scan over 35,000 GitHub directories containing `SKILL.md` and construct 150 repair tasks by injecting script faults into 100 curated packages. The Controlled Repair track contains 200 tasks from 50 packages, each provided in clean, documentation-fault, script-fault, and joint-fault states with its maintenance request held fixed. Together, the two tracks cover varied package structures, programming languages, and maintenance requirements while supporting controlled comparisons of repair and preservation. Figure 1 summarizes the benchmark and revision framework.

Across four LLMs, methods that revise both documentation and scripts can repair script faults but do not consistently outperform Markdown-only revision on documentation repair or clean-package preservation. To better coordinate repair and preservation, we introduce AST-Guided Skill Revision as an additional stage for existing revision methods. It analyzes the original package and an initial revision, links maintenance requirements to abstract syntax tree (AST) nodes, and uses calling relationships to provide context for targeted edits. Script edits are restricted to selected nodes and guided by preservation constraints derived from requirements assessed as satisfied. The documentation is then aligned with the maintenance request and revised implementation. Averaged across the four models, this stage increases repair success on faulty packages from 54.8% to 76.7% for Raw Package and from 48.8% to 76.5% for our CoEvoSkills adaptation. The proportions of tasks solved in all three runs increase by 20.8 and 31.5 percentage points, respectively, and both augmented methods improve clean-package preservation relative to their corresponding baselines.

Contributions. Our contributions are threefold.

1. A benchmark for executable skill self-evolution. We introduce SKILLSCRIPTBENCH, a 350-task benchmark that treats existing Agent Skill packages as the unit of evolution, covering documentation, scripts, and their consistency through repository-sourced tasks and matched clean and faulty states.

2. Empirical findings on repair and preservation. Across four LLMs, we show that full-package revision supports script repair but does not consistently improve documentation repair or clean-package preservation over Markdown-only revision.

3. A structure-guided self-evolution method. AST-Guided Skill Revision combines targeted script edits, preservation constraints, and documentation alignment, improving repair and preservation when applied to Raw Package and adapted CoEvoSkills outputs.

2 SKILLSCRIPTBENCH

SKILLSCRIPTBENCH evaluates how Agent Skills revise their Markdown instructions and executable scripts to satisfy maintenance requests, using 350 tasks across two complementary tracks. In-the-Wild Repair covers diverse public packages and maintenance requirements. Controlled Repair uses matched clean and faulty package states to assess documentation repair, script repair, joint repair, and preservation of correct behavior. Table 1 compares the package-maintenance coverage of SKILLSCRIPTBENCH and representative Agent Skill benchmarks (Li et al., 2026; Han et al., 2026; Zhong et al., 2026; Zhou et al., 2026).

Table 1: Comparison of Agent Skill benchmarks.
✓ / × indicate whether each dimension is explicitly covered.

Benchmark	Bundled Scripts	Skill Refinement	Doc–Code Consistency	Repair & Preservation
SkillsBench	✓	×	×	×
SWE-Skills-Bench	×	×	×	×
SkillLearnBench	×	✓	×	×
SkillGenBench	✓	✓	×	×
SkillScriptBench	✓	✓	✓	✓

2.1 TASK FORMULATION

Each SKILLSCRIPTBENCH task provides an existing executable Agent Skill package S and a natural-language request q describing the required functionality or a failure to be resolved. Revision targets the natural-language instructions and behavioral specifications in `SKILL.md`, the implementation code of bundled executable scripts, or both. A method returns a candidate package S' whose executable behavior satisfies q and whose documentation consistently describes that behavior, while preserving behavior outside the requested scope.

2.2 BENCHMARK CONSTRUCTION

We survey 35,554 GitHub-hosted Skill roots and identify 4,981 containing source files under `scripts/` or `script/`. We select packages that permit redistribution and whose behavior can be checked using repository tests or task-specific executable tests. We record the source repository and commit for each selected package.

In-the-Wild Repair. We construct 150 repair tasks from 100 packages by introducing script faults based on fault patterns identified in our GitHub survey. Each task pairs the faulty package with a maintenance request describing the expected behavior or a failure symptom. Executable checks distinguish the faulty version from a correct reference implementation. Tasks from the same package target different maintenance requirements.

Controlled Repair. This track tests whether methods repair faulty documentation and scripts while preserving correct ones. For each of 50 repository-derived packages, we keep the maintenance request fixed and construct four states: Clean without faults, Doc with documentation faults only, Script with script faults only, and Joint with both. The 200 tasks support within-package comparisons.

Quality control. We check package integrity and alignment between maintenance requests and tested behavior. Reference packages must pass the task-specific checks, while faulty versions must fail checks for the injected faults (Jia & Harman, 2011). For Controlled Repair, we verify that Doc and Script retain the unaffected artifact and that Joint combines their faults under the same request. Appendix A.1 details these construction checks.

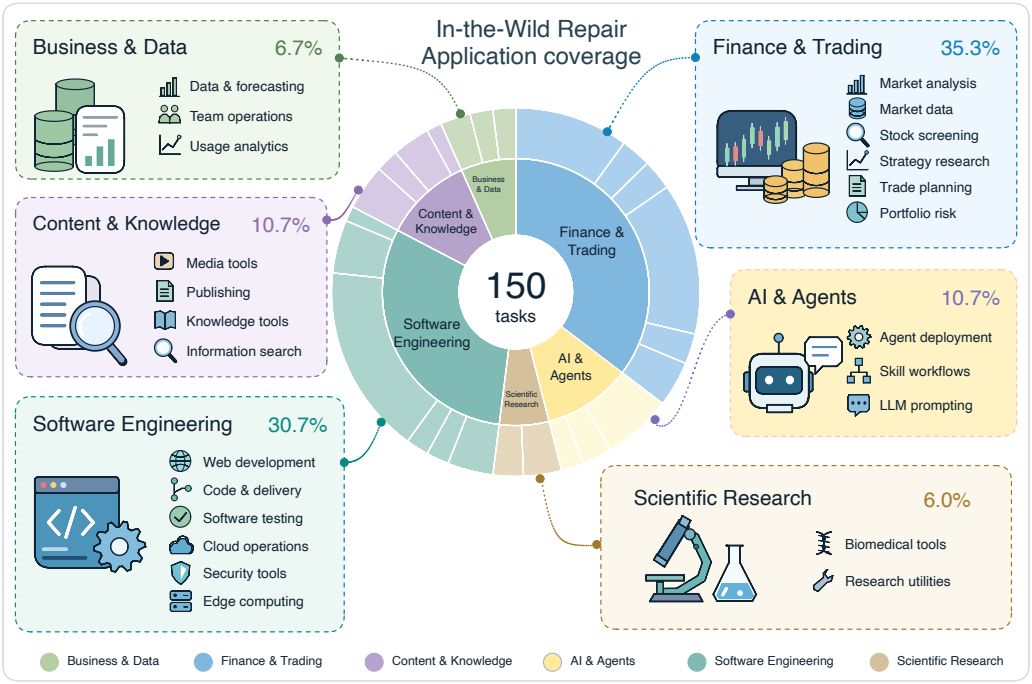

Figure 2: Application coverage of In-the-Wild Repair. Inner and outer rings show six application domains and 24 uses, respectively. Uses are listed clockwise within each domain.

2.3 Benchmark composition

Figure 2 summarizes application coverage; Appendix Figure 6 compares language and composite annotation scores across the two tracks. In-the-Wild spans six application domains and is predominantly Python, whereas Controlled Repair has a more even distribution across Python, JavaScript, and Shell, alongside mixed-language packages.

To support fine-grained analysis, we annotate tasks by maintenance requirement and programming language, and group In-the-Wild tasks by request localization. Controlled Repair comprises three diagnostic partitions: Single-Target Repair targets individual script locations, Multi-Target Repair requires coordinated changes across multiple script locations, and Cross-Call Repair targets dependencies between callers and helper functions. Appendix A.1 details the annotations and task composition.

2.4 Evaluation framework

We evaluate functional correctness and documentation–script consistency. Behavioral tests check the requested functionality and preservation of correct behavior, while documentation-driven tests execute task-relevant invocations and workflows from `SKILL.md` against the revised scripts. Success requires passing all applicable tests in both categories.

Behavioral tests check task-specific outputs, side effects, and argument propagation. Documentation-driven tests use controlled inputs and simulate external services where needed. For Clean and Doc-fault tasks, preservation checks verify that the revised scripts retain the package's existing correct functionality. All methods are evaluated against the same task-specific criteria. Appendix A.2 details the execution protocols and evaluator calibration.

3 AST-GUIDED SKILL REVISION

AST-Guided Skill Revision coordinates Markdown–script repair by linking maintenance requirements to executable structure. We assess documentation and implementation separately against the same request, identifying discrepancies to repair and correct behavior to preserve. Given an initial revision $\widetilde{S} = M(q, S)$ from a base method M, we analyze the original package S and $\widetilde{S}$ to derive repair targets and preservation constraints. These guide subsequent script edits, after which we align documented usage with q and the revised implementation to produce the final package S' (Figure 3).

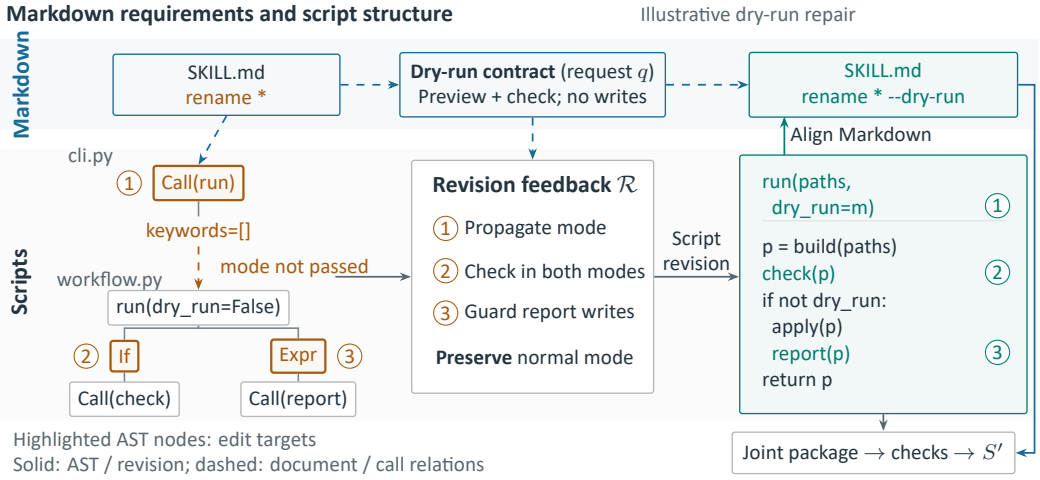

Figure 3: AST-Guided Skill Revision on a dry-run example. Numbered issues (1–3) map through revision feedback $\mathcal{R}$ to corresponding script edits, enabling preview generation and conflict checking without renaming files or writing reports in dry-run mode. The `SKILL.md` command is updated with `--dry-run` to match the repaired implementation.

3.1 REQUIREMENT–STRUCTURE GROUNDING

The LLM interprets the behavior required by q and identifies related claims in Markdown. It links these requirements and claims to their applicability conditions, supporting passages, and relevant implementation sites. Language-native AST/CST analysis binds the proposed sites to uniquely matched syntax nodes. Caller–helper relationships connect these nodes to related implementation sites, providing context for requirements that span function boundaries.

3.2 REPAIR TARGETS AND PRESERVATION CONSTRAINTS

For each grounded requirement, the LLM assesses the implementation and documentation separately under its applicability conditions. We map assessed implementation discrepancies to resolved, editable syntax nodes, forming repair targets that define the scope of subsequent script edits. Behavior assessed as satisfying the requirement supplies preservation constraints, including when the corresponding documentation requires correction. Related callers and helpers provide supporting context without necessarily becoming edit targets. We assess both S and $\widetilde{S}$, retaining requirement-linked symbols and call context from S when the initial revision removes them. Together, these assessments and structural bindings form the revision feedback $\mathcal{R} = G(q, S, \widetilde{S})$ supplied to the script editor.

3.3 COORDINATED MARKDOWN–SCRIPT REVISION

The revision model uses the repair targets, caller–helper context, and preservation constraints in $\mathcal{R}$ to generate node-level script replacements for each package version. Edits leave source text outside the selected spans unchanged and are accepted only if the resulting scripts pass syntax and applicable interface-preservation checks. Assembly uses $\widetilde{S}$ as its base, retaining its validated edits and incorporating compatible edits from S through unique source matching. The model then aligns documented

Table 2: Repair success on SKILLSCRIPTBENCH across four LLM backbones (%).

	In-the-Wild			Controlled Repair									Overall		
				Doc.			Script			Joint					
Method	Avg	P@3	Hit^3	Avg	P@3	Hit^3	Avg	P@3	Hit^3	Avg	P@3	Hit^3	Avg	P@3	Hit^3
GPT-5.5															
MD-only	0.0	0.0	0.0	26.0	42.0	12.0	0.0	0.0	0.0	0.0	0.0	0.0	4.3	7.0	2.0
Raw	53.1	76.7	27.3	30.7	44.0	18.0	45.3	48.0	42.0	48.7	52.0	46.0	47.3	62.3	31.3
Raw+AST	69.8	88.0	49.3	**72.0**	**88.0**	**52.0**	74.7	82.0	66.0	70.7	80.0	60.0	71.1	85.7	54.3
CoEvo	50.0	72.7	26.0	31.3	50.0	16.0	43.3	70.0	18.0	52.0	80.0	20.0	46.1	69.7	22.0
CoEvo+AST	**74.7**	**90.7**	**54.7**	64.7	82.0	48.0	**80.7**	**92.0**	**70.0**	**79.3**	**90.0**	**68.0**	**74.8**	**89.3**	**58.3**
GPT-5.6-Sol															
MD-only	0.0	0.0	0.0	46.7	60.0	28.0	0.0	0.0	0.0	0.0	0.0	0.0	7.8	10.0	4.7
Raw	68.2	84.7	50.0	38.7	60.0	22.0	64.7	74.0	52.0	62.7	72.0	50.0	61.8	76.7	45.7
Raw+AST	80.7	90.7	**67.3**	**75.3**	**92.0**	**52.0**	83.3	**90.0**	72.0	81.3	**90.0**	68.0	80.3	**90.7**	65.7
CoEvo	64.2	80.0	41.3	39.3	64.0	12.0	74.0	86.0	60.0	70.0	84.0	50.0	62.7	79.0	41.0
CoEvo+AST	**81.6**	**93.3**	66.0	70.0	86.0	**52.0**	**86.7**	**90.0**	**82.0**	**86.0**	88.0	**82.0**	**81.2**	**90.7**	**69.0**
Claude Opus 5															
MD-only	0.0	0.0	0.0	23.3	28.0	20.0	0.0	0.0	0.0	0.0	0.0	0.0	3.9	4.7	3.3
Raw	66.4	77.3	55.3	26.7	32.0	22.0	50.7	56.0	46.0	57.3	60.0	54.0	55.7	63.3	48.0
Raw+AST	**80.9**	**90.7**	**68.7**	66.0	**82.0**	48.0	**78.7**	82.0	**76.0**	**83.3**	86.0	**80.0**	**78.4**	**87.0**	**68.3**
CoEvo	44.4	50.7	36.7	8.7	10.0	8.0	46.0	56.0	36.0	52.7	60.0	46.0	40.1	46.3	33.3
CoEvo+AST	74.0	78.7	**68.7**	**72.0**	80.0	**58.0**	76.7	**90.0**	66.0	76.7	**90.0**	64.0	74.6	82.7	65.7
DeepSeek V4 Pro															
MD-only	0.0	0.0	0.0	12.0	12.0	12.0	0.0	0.0	0.0	0.0	0.0	0.0	2.0	2.0	2.0
Raw	61.1	64.0	57.3	13.3	14.0	12.0	66.0	68.0	64.0	63.3	64.0	62.0	54.3	56.3	51.7
Raw+AST	**78.2**	**81.3**	**74.7**	64.0	76.0	48.0	82.0	84.0	80.0	80.0	82.0	**78.0**	**76.8**	**81.0**	**71.7**
CoEvo	50.7	56.7	44.7	13.3	20.0	8.0	54.0	58.0	50.0	58.7	66.0	50.0	46.3	52.3	40.3
CoEvo+AST	70.9	75.3	66.7	**74.7**	**90.0**	**58.0**	**84.0**	**86.0**	**82.0**	**81.3**	**84.0**	**78.0**	75.4	**81.0**	69.7

MD-only, Raw, and CoEvo denote Markdown-only, Raw Package, and CoEvoSkills. Avg reports the mean success rate over three runs; P@3 and Hit^3 report the fractions of tasks successful in at least one and all three runs, respectively. Doc., Script, and Joint denote documentation, script, and joint faults. Overall pools 150 faulty-package tasks from each track. Figure 4 reports preservation on the 50 Clean tasks. Grey rows mark +AST methods; bold indicates column maxima per model, including ties.

invocations and operational instructions with the maintenance request q and the resulting scripts, correcting identified inconsistencies through localized Markdown edits supported by script evidence. Appendices A.3 and A.4 detail the feedback representation, edit validation, and package assembly rules.

3.4 INTEGRATION WITH COEVOSKILLS

We adapt CoEvoSkills (Zhang et al., 2026b) to maintenance-request-driven revision of executable Skill packages. An independent surrogate verifier assesses the package against the request and produces requirement-linked checks and a diagnosis. A failure reported by the verifier or its executed checks triggers the generator to revise `SKILL.md` and scripts using the diagnosis and check results. To obtain CoEvoSkills + AST, we use the returned package as the initial revision $\widetilde{S}$ and apply AST-Guided Skill Revision to $(q, S, \widetilde{S})$.

4 EXPERIMENTS

4.1 EXPERIMENTAL SETUP

Models and benchmark scope. We evaluate all methods with GPT-5.5, GPT-5.6-Sol, Claude Opus 5, and DeepSeek V4 Pro on all 350 SkillScriptBench tasks: 150 In-the-Wild repair cases and 200 Controlled Repair tasks. Evaluation follows the protocol in Section 2.4.

Compared methods. We compare five settings, each given the task-specific maintenance request. *Markdown-only* receives Markdown inputs and edits only `SKILL.md`. *Raw Package* and *CoEvoSkills* receive the full package and can edit `SKILL.md` and scripts; CoEvoSkills uses surrogate-verifier feedback to guide package revision, following the adaptation in Section 3.4. We augment Raw Package and CoEvoSkills with AST-Guided Skill Revision for coordinated Markdown–script repair, obtaining *Raw Package + AST* and *CoEvoSkills + AST*, respectively. Appendix A.5 details the model configurations, revision protocols, and prompt templates.

Repetitions and metrics. We report three runs for each model–method–task combination. Avg (Pass@1) is the mean success rate across runs; Pass@3 (P@3) and Hit^3 are the fractions of tasks solved in at least one run and all three runs, respectively (Chen et al., 2021; Yao et al., 2024). Table 2 reports repair success, with Overall pooling the 300 faulty-package tasks. Figure 4 summarizes preservation on the 50 Clean tasks; Appendix Table 18 gives per-model results. Cross-model means weight the four models equally. Appendix Table 13 additionally reports P@3 on Clean tasks and the full 350-task benchmark. We report paired source-package bootstrap confidence intervals (Davison & Hinkley, 1997) for Overall Avg gains in Appendix Table 14.

4.2 Main Results

Full-package baselines achieve broader repair coverage, but documentation repair remains a weakness. Averaged across four backbones, Raw Package and CoEvoSkills achieve Overall Avg scores of 54.8% and 48.8%, respectively. Both baselines repair script faults, joint faults, and In-the-Wild cases. However, their documentation-repair Avg scores are 27.3% and 23.2%, respectively, compared with 27.0% for Markdown-only.

AST-guided revision improves repair success across all four backbones. Raw Package + AST and CoEvoSkills + AST achieve Overall Avg scores of 76.7% and 76.5%, improving over their respective baselines by 21.9 and 27.7 pp. Their P@3 reaches 86.1% and 85.9%, while Hit^3 reaches 65.0% and 65.7%, respectively. All three metrics improve for both generators on every backbone.

5 Analysis

We analyze package revision along two dimensions: the balance between fault correction and behavior preservation, and the variation in revision gains across task characteristics.

5.1 RQ1: What does AST-guided revision add beyond full-package access?

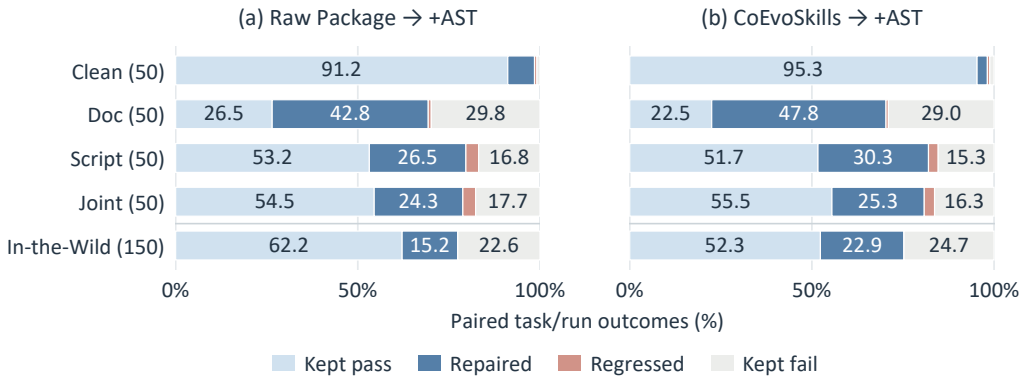

Figure 4: AST-guided revision improves repair and preservation. Paired outcomes for (a) Raw Package and (b) CoEvoSkills before and after AST-guided revision, pooled across four backbones and three runs. Parentheses indicate task counts.

Broader repair coverage does not imply better preservation of correct behavior. As reported in Appendix Table 19, Raw Package and CoEvoSkills achieve lower Avg on Clean tasks than

Markdown-only across all four backbones, with mean gaps of 7.8 and 3.7 percentage points, respectively. These results highlight the need to distinguish what must change from what should remain intact during package revision.

AST-guided revision improves fault correction and behavior preservation together. By linking Markdown requirements to script structure, its feedback provides both repair targets and preservation constraints. As shown in Figure 4, AST-guided revision corrects substantially more failures than it introduces. Applied to Raw Package on documentation-fault tasks, it changes 42.8% of task/run pairs from failure to success, whereas only 0.8% change from success to failure. Corrections also outnumber regressions on script and joint faults for both generators. On Clean tasks, mean Hit^3 rises from 85.5% to 96.5% for Raw Package and from 93.0% to 96.5% for CoEvoSkills, indicating more consistent preservation across all three runs.

In the ablation study on Controlled Repair with GPT-5.6-Sol, *w/o Structural Feedback* and *w/o Call Context* retain Markdown alignment and achieve 69.2% and 67.3% Avg, respectively, compared with 84.8% for AST-Guided Skill Revision. The full configuration outperforms both variants across all three fault states and in Clean-state preservation, with the largest margins on Joint tasks. Appendix A.6 provides configuration details and state-wise results.

5.2 RQ2: How do revision gains vary across task characteristics?

We compare Raw Package with Raw Package + AST on faulty-package tasks, examining structural partitions in Controlled Repair and repair categories, programming languages, and request localization in In-the-Wild.

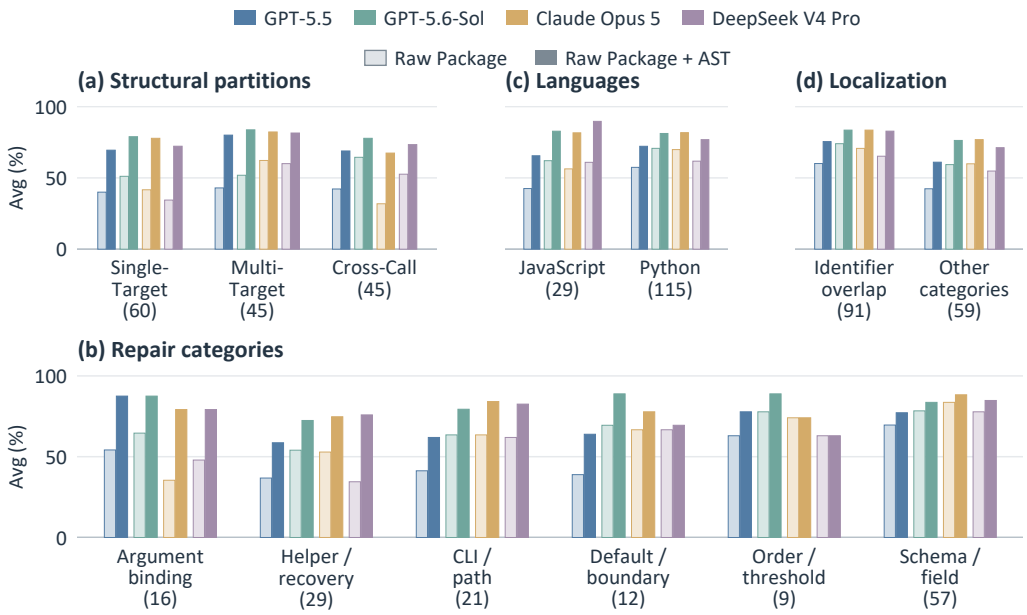

Figure 5: Revision gains vary across task characteristics. Panel (a) uses faulty-package tasks from Controlled Repair; panels (b–d) use In-the-Wild. Light/dark bars show Raw Package/Raw Package + AST Avg over three runs for each model. Parentheses give task counts; complete repair-category results appear in Appendix Figure 7.

Structural partitions. AST-guided revision improves both single-target and coordinated repairs across all four backbones. As shown in Figure 5(a), mean Avg increases by 32.8, 27.6, and 24.1 percentage points on Single-Target, Multi-Target, and Cross-Call, respectively. Multi-Target reaches the highest post-revision Avg of 81.9% and outperforms Single-Target across all four backbones, demonstrating that the benefits extend beyond isolated edits. Cross-Call improves to 71.9% but remains the least successful partition.

Repair categories. AST-guided revision yields its largest gains on argument-binding and helper/recovery repairs. As shown in Figure 5(b), their Avg rises from 50.5% to 83.3% and from 44.5% to 70.4%, respectively. Argument-binding repairs thus reach nearly the same success rate as schema/field mapping, which improves from a higher baseline of 77.3% to 83.5%. Avg increases from 27.8% to 30.6% on the three branch/route tasks and remains at 61.1% on the three multi-file pipeline tasks.

Programming languages. AST-guided revision produces larger gains on JavaScript than Python across all four backbones. Figure 5(c) shows that cross-model mean Avg rises from 55.5% to 79.9% for JavaScript and from 64.9% to 78.0% for Python. The two language groups thus reach similar post-revision success rates despite JavaScript's substantially lower baseline.

Request localization. AST-guided revision reduces the performance gap across request-localization categories. As shown in Figure 5(d), Avg improves by 13.8 percentage points for identifier-overlap tasks and 17.2 for the remaining tasks, narrowing the cross-model mean gap from 13.4 to 10.0 points.

6 Related Work

Agent Skill Self-Evolution. SkillOpt and SkillAudit focus on revising skill documents, using bounded textual edits and paired-trajectory feedback, respectively (Yang et al., 2026; Gao et al., 2026). CoEvoSkills and SkillEvolver support broader package-level evolution, jointly constructing and refining natural-language instructions and executable code (Zhang et al., 2026b;a). However, downstream task success alone does not separately assess documentation repair, script repair, documentation–script consistency, and preservation of correct behavior.

Benchmarks for Agent Skills. SkillsBench and SWE-Skills-Bench evaluate the utility of supplied skills across diverse agent tasks and repository-based software engineering, respectively (Li et al., 2026; Han et al., 2026). SkillLearnBench evaluates continual skill generation through specification quality, execution trajectories, and task outcomes, while SkillGenBench compares pipelines that distill repositories and documents into reusable skills (Zhong et al., 2026; Zhou et al., 2026). However, these benchmarks do not systematically evaluate the repair of existing Skill packages comprising both Markdown instructions and executable scripts.

Program Repair. Structure-aware program repair uses program representations to guide code changes. Astor applies AST-level transformations to Java programs, Getafix learns hierarchical fix patterns from past edits, and TBar applies repair templates at localized fault sites (Martinez & Monperrus, 2016; Bader et al., 2019; Liu et al., 2019). Repository-level benchmarks such as SWE-bench evaluate patches against real issue descriptions and executable tests (Jimenez et al., 2024). These settings do not explicitly address joint revision of Markdown instructions and executable scripts to restore their consistency.

7 Conclusion

We introduced SkillScriptBench, a 350-task benchmark for self-evolution of executable Agent Skill packages. It combines In-the-Wild Repair, built from a scan of over 35,000 public Skill roots, with Controlled Repair across matched clean, documentation-fault, script-fault, and joint-fault states. Across four LLM backbones, full-package baselines broaden repair coverage but do not consistently outperform Markdown-only revision in documentation repair or clean-package preservation. We proposed AST-Guided Skill Revision to coordinate Markdown and script revisions through requirement-grounded structural feedback. Applied to Raw Package and CoEvoSkills, it improves both repair and preservation while increasing mean all-three-run success (Hit^3) across all four artifact states. Evaluation of executable Skill evolution should distinguish fault repair from preservation of correct package behavior.

AI USE STATEMENT

Generative AI tools assisted with literature search and source organization, outline and argument development, language drafting, and LaTeX preparation. They also assisted in organizing and packaging the code, benchmark inputs, and experimental results for the anonymous repository. The authors reviewed the references, numerical claims, manuscript text, and figures against the source materials and experimental results. The authors take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

ETHICS STATEMENT

The study uses public software artifacts and model-generated package revisions. The release will include only artifacts permitted for redistribution, with source attribution. Construction labels and evaluators are kept separate from method inputs to reduce evaluation leakage.

REPRODUCIBILITY STATEMENT

The anonymous repository at `https://anonymous.4open.science/r/SkillScriptBench-3A8C/` contains the revision code and prompt templates, the task inputs underlying the main results, and per-run outcomes with scripts that reproduce Table 2. Sections 2–5 describe benchmark construction, the method, and evaluation. Appendix A details evaluation checks and experimental settings; Appendices B and C provide method and baseline prompts.

REFERENCES

Agent Skills. Agent Skills specification. Online specification, 2026. URL `https://agentskills.io/specification`. Accessed 2026-08-27.

Babel Contributors. @babel/parser. Babel documentation, n.d. URL `https://babeljs.io/docs/babel-parser`. Accessed 2026-09-15.

Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. Getafix: Learning to fix bugs automatically. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):159:1–159:27, 2019. doi: 10.1145/3360585. URL `https://doi.org/10.1145/3360585`.

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, 2021. URL `https://arxiv.org/abs/2107.03374`.

A. C. Davison and D. V. Hinkley. *Bootstrap Methods and their Application*. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, 1997. doi: 10.1017/CBO9780511802843. URL `https://doi.org/10.1017/CBO9780511802843`.

Haowen Gao, Haoran Chen, Can Wang, Shasha Guo, Liang Pang, Zhaoyang Liu, Huawei Shen, and Xueqi Cheng. SkillAudit: Ground-truth-free skill evolution via paired trajectory auditing, 2026. URL `https://arxiv.org/abs/2606.14239`.

Tingxu Han, Yi Zhang, Wei Song, Chunrong Fang, Zhenyu Chen, Youcheng Sun, and Lijie Hu. SWE-Skills-Bench: Do Agent Skills actually help in real-world software engineering?, 2026. URL `https://arxiv.org/abs/2603.15401`.

Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011. doi: 10.1109/TSE.2010.62. URL https://doi.org/10.1109/TSE.2010.62.

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. URL https://openreview.net/forum?id=VTF8yNQM66.

Xiangyi Li, Wenbo Chen, Yimin Liu, Shenghan Zheng, Xiaokun Chen, Yifeng He, Yubo Li, Bingran You, Haotian Shen, Jiankai Sun, Shuyi Wang, Qunhong Zeng, Di Wang, Xuandong Zhao, Yuanli Wang, Roey Ben Chaim, Zonglin Di, Yipeng Gao, Junwei He, Yizhuo He, Liqiang Jing, Luyang Kong, Xin Lan, Jiachen Li, Songlin Li, Yijiang Li, Yueqian Lin, Xinyi Liu, Xuanqing Liu, Haoran Lyu, Ze Ma, Bowei Wang, Runhui Wang, Tianyu Wang, Wengao Ye, Yue Zhang, Hanwen Xing, Yiqi Xue, Steven Dillmann, and Han-chung Lee. SkillsBench: Benchmarking how well Agent Skills work across diverse tasks, 2026. URL https://arxiv.org/abs/2602.12670.

Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. TBar: Revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 31–42, 2019. doi: 10.1145/3293882.3330577. URL https://doi.org/10.1145/3293882.3330577.

Yuxuan Liu, Zhaochen Su, Lingyun Xie, Yuhao Zhang, Qing Zong, Jiahe Guo, Zhongwei Xie, Yiyan Ji, Yauwai Yim, Hongyu Luo, Xiyu Ren, Chenyu Ruan, Haoran Li, and Yangqiu Song. SkillRevise: Improving LLM-authored Agent Skills via trace-conditioned skill revision, 2026a. URL https://arxiv.org/abs/2606.01139. Accepted to Findings of EMNLP 2026.

Yuxuan Liu, Zhaochen Su, Yuhao Zhang, Jiahe Guo, Zhongwei Xie, Huihao Jing, Lingyun Xie, Qing Zong, Yauwai Yim, Zhixiong Zhang, Haoran Li, and Yangqiu Song. Rethinking self-evolving Agent Skills: Feedback dynamics over multiple rounds, 2026b. URL https://arxiv.org/abs/2608.02636.

Matias Martinez and Martin Monperrus. ASTOR: A program repair library for Java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pp. 955–958, 2016. doi: 10.1145/2931037.2948705. URL https://doi.org/10.1145/2931037.2948705.

Python Software Foundation. ast: Abstract syntax trees. Python documentation, n.d. URL https://docs.python.org/3/library/ast.html. Accessed 2026-09-15.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, 2023. doi: 10.52202/075280-0377. URL https://arxiv.org/abs/2303.11366.

Tree-sitter Contributors. Tree-sitter: Introduction. Tree-sitter documentation, n.d. URL https://tree-sitter.github.io/tree-sitter/. Accessed 2026-09-15.

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, 2023. URL https://arxiv.org/abs/2305.16291.

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. doi: 10.52202/079017-1601. URL https://arxiv.org/abs/2405.15793.

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, 2026. URL https://arxiv.org/abs/2605.23904.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023. doi: 10.48550/arXiv.2210.03629. URL https://arxiv.org/abs/2210.03629.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains, 2024. URL https://arxiv.org/abs/2406.12045.

Genrui Zhang, Erle Zhu, Jinfeng Zhou, Caiyan Jia, and Hongning Wang. SkillEvolver: Skill learning as a meta-skill, 2026a. URL https://arxiv.org/abs/2605.10500.

Hanrong Zhang, Shicheng Fan, Henry Peng Zou, Yankai Chen, Zhenting Wang, Jiayu Zhou, Chengze Li, Wei-Chieh Huang, Yifei Yao, Kening Zheng, Xue Liu, Xiaoxiao Li, and Philip S. Yu. CoEvoSkills: Self-evolving Agent Skills via co-evolutionary verification, 2026b. URL https://arxiv.org/abs/2604.01687. COLM 2026.

Shanshan Zhong, Yi Lu, Jingjie Ning, Yibing Wan, Lihan Feng, Yuyi Ao, Leonardo F. R. Ribeiro, Markus Dreyer, Sean Ammirati, and Chenyan Xiong. SkillLearnBench: Benchmarking continual learning methods for Agent Skill generation on real-world tasks, 2026. URL https://arxiv.org/abs/2604.20087. COLM 2026.

Yifan Zhou, Zhentao Zhang, Ziming Cheng, Shuo Zhang, Qizhen Lan, Zhangquan Chen, Zhi Yang, Qianyu Xu, Ronghao Chen, Huacan Wang, and Sen Hu. SkillGenBench: Benchmarking skill generation pipelines for LLM agents, 2026. URL https://arxiv.org/abs/2605.18693.

A BENCHMARK, METHOD, AND EVALUATION DETAILS

A.1 BENCHMARK COMPOSITION AND CONSTRUCTION CHECKS

Table 3 details the composition of the two benchmark tracks; Table 5 defines the four matched Controlled Repair states.

Table 3: Benchmark sizes, language coverage, and construction profiles. Language counts refer to repair cases for In-the-Wild and canonical base packages for Controlled Repair. TypeScript and mixed-language packages are reported separately here and grouped as Others in Figure 6.

Track	Size	Language profile	Construction profile
In-the-Wild	150 cases	115 Python; 29 JS; 3 TS; 3 Shell	Composite score: 72 at 0–2; 51 at 3–5; 27 at ≥ 6
Controlled Repair	$50 \times 4 = 200$ tasks	18 Python; 14 JS; 14 Shell; 2 TS; 2 mixed	Controlled four-state construction with three diagnostic partitions

In-the-Wild sources and case construction. Our scan identifies 35,554 Skill roots containing `SKILL.md`, of which 4,981 (14.0%) contain source files under a package-local `scripts/` or `script/` subtree. We select packages with licenses permitting redistribution, traceable source revisions, and task-relevant behavior that can be tested deterministically under controlled conditions. Repair cases are validated using repository tests or contract-derived probes that accept reference implementations and reject the corresponding faulty variants.

Table 4: Source inventory and inclusion criteria for In-the-Wild Repair. The first two rows report the source scan; the last two describe the curated repair set.

Collection	Definition or inclusion criterion	Count
Skill roots	Contains `SKILL.md`	35,554
Roots with source	Source files under `scripts/` or `script/`	4,981
Selected packages	Redistributable license, traceable revision, testable behavior	100
Repair cases	Reference passes and faulty variant fails task-specific checks	150

We construct repair cases by introducing source-level faults into repository-derived Skill packages through parameter substitution, helper-call bypassing, and changes to default-value handling. Each case pairs the faulty package with a maintenance request specifying the required behavior or describing a failure symptom. Cases derived from the same package target distinct behavioral requirements. The 150 cases span eight repair categories, with their distribution reported in Figure 7. The case registry records source repositories, commits, package identities, and content hashes. Fault and state labels, reference packages, and benchmark acceptance checks are reserved for construction and evaluation and excluded from method inputs.

Application coverage. Figure 2 shows six application domains and their 24 uses, classified by the source packages' documented purpose. Counts refer to repair tasks; cases derived from the same package share its application label.

Language and composite annotations. Figure 6 compares the two tracks by programming language and composite annotation score. The score sums indicators for target count (one point for at least two targets and two additional points for at least four), target-file count (two points for at least two files), call-related metadata (two points), the recorded `symptom_based` flag (one point), package size (one point for at least ten files), and script count (one point for at least three files). Scores are grouped as 0–2, 3–5, and ≥ 6. For Controlled Repair, the request flag comes from construction metadata or request-text rules; for In-the-Wild, it marks tasks outside the identifier-overlap category.

Request localization. For In-the-Wild, request localization is labeled by matching the request against package code-file paths, filenames, and extracted identifiers, in that order. The identifier-overlap category contains 91 tasks; the other categories comprise 12 path matches, one filename match, and 46 tasks with no detected match.

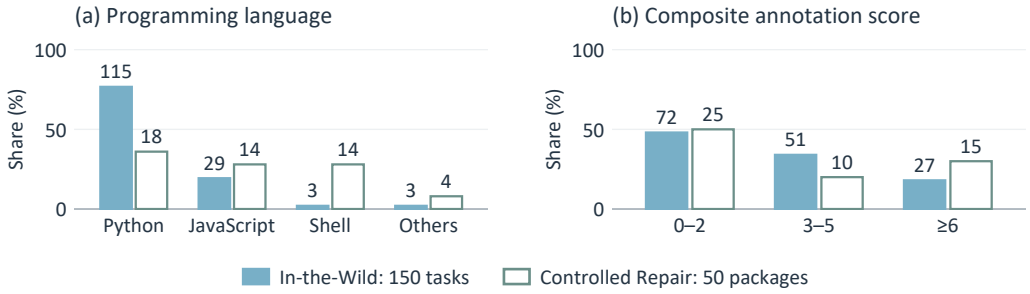

Figure 6: Language and composite annotation scores of the two benchmark tracks. Counts refer to 150 In-the-Wild tasks and 50 Controlled Repair base packages. Bars show percentages; labels give counts. Others combines TypeScript and mixed-language packages.

Controlled Repair bases. Controlled Repair uses 50 canonical Skill packages derived from repository sources. Construction records document their source revisions, licenses, and injected faults. Each package pairs a reference implementation with a description of the behavior targeted by its maintenance request. For Multi-Target and Cross-Call, this behavior is specified in an operational-contract block in `SKILL.md`. We construct four states from each canonical package while holding the maintenance request fixed: Clean retains the canonical package, whereas Doc., Script, and Joint introduce faults into Markdown, scripts, or both, respectively (Table 5).

Table 5: Construction of the four matched Controlled Repair states. Faults are injected into Markdown, scripts, or both, while the maintenance request is held fixed within each package group. Unchanged denotes the corresponding artifact from the canonical package before revision.

State	Markdown	Scripts	Evaluation focus
Clean	Unchanged	Unchanged	Preservation
Doc.	Fault-injected	Unchanged	Documentation repair
Script	Unchanged	Fault-injected	Script repair
Joint	Fault-injected	Fault-injected	Joint repair

Appendix A.8 instantiates these four states for the VCP screener and follows a Joint task through revision and execution.

Table 6: Composition of the three Controlled Repair partitions. Partitions distinguish how repair targets are organized. Language and composite score counts refer to base packages, each contributing four tasks with different artifact states.

Partition	Bases/tasks	Language profile	Score 0–2	Score 3–5	Score ≥ 6
Single-Target	20/80	9 JS; 9 Shell; 2 TS	20	0	0
Multi-Target	15/60	5 JS; 5 Shell; 3 Python; 2 mixed	5	4	6
Cross-Call	15/60	15 Python	0	6	9

Repair targets: individual script locations (Single-Target), 2–3 script locations (Multi-Target), and 1–9 mutated call bindings (Cross-Call).

Construction checks. Static checks verify package integrity and source metadata. For Controlled Repair, we also verify that each base contributes four states with identical maintenance requests and artifact changes matching the assigned state. In each of the Multi-Target and Cross-Call partitions, all 15 canonical bases and all four reference-repaired states per base pass the private contract tests during calibration.

Request information. Each Controlled Repair base has one maintenance request shared across its four states. Of these 50 requests, 22 specify behavioral contracts only, 18 describe failure symptoms only, and 10 name an executable path or symbol associated with the repair target. Requests are checked for replacement-code fragments.

A.2 Evaluation details

For task t, $\mathcal{C}_t$ contains the acceptance checks for required behavior, documented use, and preservation of correct functionality. All methods are evaluated with the same task-specific checks. A revised package S' succeeds when all checks pass:

$$Y_t(S') = \bigwedge_{c \in \mathcal{C}_t} c(S').$$

Documentation-driven execution. Task-specific evaluators extract commands and workflows from the revised `SKILL.md` and execute them with the revised scripts under predefined test scenarios. Entry points and arguments come from the documented commands; working directories and variable bindings come from the surrounding workflow. Shell parsing preserves quoting and line continuations. Test inputs instantiate documented placeholders, and controlled fixtures supply external dependencies. Assertions check exit status, outputs, generated artifacts, and downstream arguments against the task requirements.

Documented task contracts. Task-specific rules bind documented options to their commands and test the resulting behavior against the usage requirements. For example, the VCP history workflow checks the requested stock symbol, scan and stride parameters, and the number of price bars used in the forward evaluation window. The VCP documentation verdict combines command execution, output assertions, and checks that referenced package files are present and readable. Appendix A.8 gives the executed command, test inputs, and outcomes for a VCP repair case.

Command coverage and verdicts. Command-coverage checks match task-relevant documented invocations to execution records and verify their behavior against task-specific assertions. Missing invocations, empty extraction results, and unresolved workflow context are examined to distinguish package defects from evaluator or environment errors. Evaluator or environment errors are corrected before re-evaluating the unchanged package.

Preservation. For documentation-fault tasks, evaluators execute invocations from both the revised and reference documentation against the revised scripts. The former tests the revised usage instructions; the latter tests preservation of existing behavior. Clean tasks require the revised package to retain both its behavioral correctness and documented-use correctness.

Executable behavior. Behavioral tests exercise target functions, script entry points, caller–helper workflows, or repository test suites. Function-level tests execute the selected source with its required dependency closure; entry-point probes import the revised module or launch the revised script. Assertions check task-relevant return values, emitted files, exit status, output streams, or arguments passed to downstream calls. Controlled fixtures supply external CLI tools, HTTP responses, hardware interfaces, and other dependencies as needed.

Execution environment. The entry-point probe suite runs as an unprivileged user in a network-disabled container with a read-only filesystem and a writable temporary directory. The revised package and probe code are mounted read-only. The standard container limit is one CPU, 512 MiB memory, and a 35-second outer timeout; the browser fixture uses a separate image and a 1 GiB memory limit. Other execution backends use task-specific subprocess or language-runtime boundaries. Execution logs record the package hash, command, exit status, outputs, and assertion outcomes.

Behavioral-check calibration. We calibrate behavioral checks using behavior-preserving rewrites and faulty package variants (Table 7). For each Single-Target base, the positive control uses an equivalent script rewrite, such as `400 <= prev.status` in place of `prev.status >= 400`; the negative control appends the correct fragment as a comment while retaining the script fault.

For Cross-Call, we construct two controls at each of 49 targets, modifying one call argument in a reference package: the positive variant replaces a variable `x` with `(x,)[0]`, whereas the negative variant substitutes a different caller variable.

Table 7: Behavioral-check calibration on constructed package variants. Counts give variants with the expected behavioral verdict.

Partition	Bases	Control	Recorded verdict
Single-Target	20	Equivalent script rewrite	20/20 accepted
		Comment-only false repair	20/20 rejected
Cross-Call	15	Equivalent argument expression	49/49 accepted
		Single incorrect argument binding	49/49 rejected

Discriminating test inputs. Caller–helper probes use distinct values at different argument positions so that swapping or reusing an argument produces an observable failure. For a CLI limit, probes cover an omitted option, zero, and nonzero explicit values, checking the value in the outgoing API request.

Acceptance-check coverage. Table 8 connects package requirements to executable checks, with recorded controls and the VCP revisions illustrating how the checks distinguish valid repairs from faulty outputs.

Table 8: Representative acceptance checks and recorded discriminating evidence. The script controls are summarized in Table 7; the VCP revisions are detailed in Appendix A.8.

Requirement	Executable check	Discriminating evidence
Script behavior	Execute target code and downstream argument bindings.	Accept equivalent rewrites; reject comment-only repairs and incorrect bindings.
Documented entry points	Execute the entry point and options specified by the document.	Accept a renamed vault entry point with matching documentation; reject a documented command with an unsupported option.
Task outputs	Check VCP detections, requested scan settings, and forward evaluation windows.	Raw Package misses expected detections despite exiting successfully; Raw Package + AST passes both contracting price scales and the non-contracting input.
Required artifacts	Inspect created vault directories, document contents, and verification reports against the fixture state.	Accept helper extraction and equivalent report wording; reject missing artifacts and false healthy-status reports.
Existing functionality	Check the new vault reference together with pre-existing configuration contents.	Accept the reference control; reject a variant that adds the required reference but erases unrelated configuration.

A.3 Requirement feedback and structural binding

Shared requirement records. The LLM reads q, `SKILL.md`, and the supported scripts in S and $\widetilde{S}$ to extract behavioral requirements, applicability conditions, and supporting passages (Table 9). Request passages support explicit requirements and inferences from reported problems; Markdown passages support document claims assessed against the requested behavior. The records link these requirements to implementation sites for script revision. Markdown alignment checks the same requested behavior against the assembled scripts, as described in Appendix A.4.

Table 9: Requirement feedback in $\mathcal{R}$. Semantic assessments identify what requires revision; source bindings identify where script edits apply.

Field	Content	Role
Requirement	Subject, relation, object, polarity, applicability condition, and documentation requirement	Specifies required or prohibited behavior
Evidence	Source kind, passage identifier, quotation, and semantic confidence (`high`/`low`)	Grounds the requirement and its conditions
Binding	Snapshot, file, node type, source quotation, optional symbol and line hint; `EDIT`/`CONTEXT` use	Locates repair targets and supporting code
Assessment	Per-snapshot code assessment; document evidence, condition overlap, and document-review coverage	Separates script repair, document alignment, and behavior to preserve

Code and document assessments. The LLM assesses each requirement separately in S and $\widetilde{S}$. `REPAIR` identifies an implementation discrepancy, while `SATISFIED` identifies behavior to retain. Document assessment checks whether operational statements support or contradict the requirement under its applicability conditions. A normative contradiction under the same or overlapping conditions yields `CONTRADICTED`. Supporting evidence under the same conditions, review of the full document, and an unambiguous assessment yield `SATISFIED`. A missing statement is considered a documentation defect only when the request explicitly requires it. Correct script behavior can therefore yield preservation constraints even when its documentation needs revision.

Programmatic grounding. The host assigns passage identifiers, verifies source quotations, and constructs syntax inventories independently of semantic discovery for supported files under `scripts/`: Python uses `ast` (Python Software Foundation, n.d.), JavaScript/TypeScript uses Babel (Babel Contributors, n.d.), and shell uses Tree-sitter (Tree-sitter Contributors, n.d.). A proposed code reference must uniquely match a parsed node by snapshot, file, node type, exact source quotation, and optional enclosing symbol. An exact line hint can disambiguate repeated matches. Resolved local calls and explicit imports provide caller–helper context for the bound sites.

Editing scope and preservation. A script target requires a high-confidence requirement, a `REPAIR` code assessment for its snapshot, and a uniquely resolved `EDIT` binding to an editable node. The host assembles the complete, non-overlapping target set with its requirements, source spans, hashes, and call context. High-confidence requirements with a `SATISFIED` code assessment supply preservation constraints. Requirements with an `UNCERTAIN` code assessment supply neither edit permissions nor preservation constraints for that snapshot. `CONTEXT` bindings supply supporting structure only. Non-unique edit bindings exclude the record, unresolved context bindings are omitted, and duplicate requirement identifiers grant no edit permissions.

A.4 Unified revision and package assembly

Unified configuration. AST-Guided Skill Revision applies the same discovery, editing, budget, and assembly rules to every task, with parsers selected by source language. Given q, S, and the initial revision $\widetilde{S}$, a single discovery response provides requirement records for both snapshots. The editor receives the request, source snapshot, admitted targets, caller–helper context, and preservation constraints. Target admission follows Appendix A.3; Table 10 gives the stage budgets.

Table 10: Stage budgets for AST-guided revision after the initial generator output. The procedure performs one pass through these stages.

Stage	Model-call budget	Edit budget
Requirement discovery	One, covering S and $\widetilde{S}$	No edits
Script revision	At most one per snapshot with admitted targets	At most eight non-overlapping nodes per snapshot; at most 4,096 UTF-8 bytes per replacement
Markdown alignment	One, after script assembly	At most eight non-overlapping line-addressed edits to `SKILL.md`

Script editing interface. The native editor either abstains for a branch or returns a node identifier and a non-empty, source-changing replacement for every admitted target exactly once. Source hashes bind the response to its input snapshot. Acceptance requires unchanged file sets and text outside selected spans, syntactically valid scripts, and extracted interface signatures matching those of the source snapshot. Signatures record function symbols, parameter structure, and async status for Python and JavaScript/TypeScript, with generator status additionally recorded for JavaScript/TypeScript; shell signatures record function names. Over-budget or overlapping target sets are rejected as a whole, and invalid responses leave the branch unavailable for assembly.

Script assembly. Assembly starts from the structurally validated revision of $\widetilde{S}$ when available, or $\widetilde{S}$ otherwise, and incorporates validated edits from S. Each edit requires a unique match to its original source block or, if absent, to its replacement, indicating an already-applied edit. The full edit set is applied only when every edit is accounted for and the assembled package passes syntax checks and preserves the interface signatures of the starting package; otherwise the starting package is retained unchanged.

Markdown alignment. After script assembly, the LLM aligns `SKILL.md` with q and the assembled implementation, including when no script edits were made. Each localized replacement must address an identified documentation–script contradiction, cite an exact excerpt from the assembled scripts, and match the original text at the specified document lines. Replacements must be non-empty and change the original content. Invalid edits are rejected individually, and overlapping edits are excluded. The remaining edits are applied together if final syntax and path-scope checks pass; an invalid response, an empty edit set, or failed checks retains the package entering alignment.

Validation and output. Final checks verify script syntax and restrict changed paths to `SKILL.md` and files under `scripts/`. The resulting package is S'; the procedure ends after one pass of script revision and Markdown alignment.

A.5 Compared methods and additional results

Model configuration. Table 11 lists the API model identifiers corresponding to the four backbones. For the four-backbone comparison, the request adapters set `temperature` to 0 and leave `top_p` and `reasoning_effort` unspecified.

Table 11: Model names and API identifiers.

Model	API identifier
GPT-5.5	`gpt-5.5`
GPT-5.6-Sol	`gpt-5.6-sol`
Claude Opus 5	`claude-opus-5`
DeepSeek V4 Pro	`deepseek-v4-pro`

Baseline inputs and prompts. Table 12 summarizes each method's inputs and editing scope. Markdown-only receives Markdown files and edits `SKILL.md`. Its initial patch is followed by one revision using the maintenance request, revised Markdown, and feedback on syntax validity and editing-scope compliance. Raw Package receives the public package, including scripts, and generates package edits. Full-package editing is restricted to `SKILL.md` and `scripts/`. The Markdown-only patch interface returns file paths, source hashes, line ranges, and replacement text. Task-specific acceptance tests evaluate the returned packages after revision.

CoEvoSkills adaptation. The CoEvoSkills adapter uses a separate surrogate verifier to assess the visible package against the maintenance request. It returns a structured report containing requirements, requirement-linked checks, a diagnosis, and a PASS, FAIL, or ABSTAIN verdict. The proposed checks cover file presence or absence, required or forbidden text, and script syntax. They are executed programmatically. Each run uses one surrogate-verification call. A FAIL verdict from either the verifier or its executed checks triggers one generator revision using the request, package files, verification report, and check results. The run then terminates. If neither verdict is FAIL, the input

package is retained. Both the surrogate verifier and the generator use a sampling temperature of 0, without an explicitly specified output-token limit.

Appendix C reproduces the system and user prompt templates for all three baselines.

Table 12: Method inputs, editable artifacts, and revision protocols. No evolution provides a calibration reference; +AST augments the corresponding generator.

Approach	Visible input	Editable output	Revision protocol
No evolution	Input package	None	Return the input unchanged
MD-only	Request; Markdown	`SKILL.md`	Initial patch; one generic revision
Raw Package	Request; full package	`SKILL.md`; scripts	Package-edit generation
CoEvoSkills	Request; full package	`SKILL.md`; scripts	One verification; at most one generator revision
Raw Package + AST	Request; original and Raw packages	`SKILL.md`; scripts	Raw generation; AST-guided revision
CoEvoSkills + AST	Request; original and CoEvo packages	`SKILL.md`; scripts	CoEvo generation; AST-guided revision

For +AST, the original package S and the corresponding generator output $\widetilde{S}$ are passed to the procedure in Appendix A.4. Appendix B provides prompt templates for requirement discovery, script revision, and Markdown alignment.

The evaluation covers 350 tasks, four models, and five methods, with three runs per setting. Avg is the mean success rate across runs; Hit^3 is the fraction of tasks that pass all three runs. Table 13 reports Pass@3, the fraction of tasks solved in at least one run, and Table 15 gives the run-level counts.

Table 13: Three-run coverage (P@3) on SKILLSCRIPTBENCH (%): tasks solved in at least one run. Controlled Repair includes all four states; All tasks covers all 350 tasks. Abbreviations follow Table 2. Mean weights models equally; bold marks column maxima within each block, including ties.

Method	In-the-Wild	Controlled Repair					All tasks
		Clean	Doc.	Script	Joint	All	
GPT-5.5							
MD-only	0.0	**100.0**	42.0	0.0	0.0	35.5	20.3
Raw	76.7	96.0	44.0	48.0	52.0	60.0	67.1
Raw+AST	88.0	**100.0**	**88.0**	82.0	80.0	87.5	87.7
CoEvo	72.7	98.0	50.0	70.0	80.0	74.5	73.7
CoEvo+AST	**90.7**	**100.0**	82.0	**92.0**	**90.0**	**91.0**	**90.9**
GPT-5.6-Sol							
MD-only	0.0	**100.0**	60.0	0.0	0.0	40.0	22.9
Raw	84.7	96.0	60.0	74.0	72.0	75.5	79.4
Raw+AST	90.7	**100.0**	**92.0**	**90.0**	**90.0**	**93.0**	**92.0**
CoEvo	80.0	98.0	64.0	86.0	84.0	83.0	81.7
CoEvo+AST	**93.3**	98.0	86.0	**90.0**	88.0	90.5	91.7
Claude Opus 5							
MD-only	0.0	**100.0**	28.0	0.0	0.0	32.0	18.3
Raw	77.3	96.0	32.0	56.0	60.0	61.0	68.0
Raw+AST	**90.7**	**100.0**	**82.0**	82.0	86.0	87.5	**88.9**
CoEvo	50.7	**100.0**	10.0	56.0	60.0	56.5	54.0
CoEvo+AST	78.7	**100.0**	80.0	**90.0**	**90.0**	**90.0**	85.1
DeepSeek V4 Pro							
MD-only	0.0	**100.0**	12.0	0.0	0.0	28.0	16.0
Raw	64.0	96.0	14.0	68.0	64.0	60.5	62.0
Raw+AST	**81.3**	**100.0**	76.0	84.0	82.0	85.5	**83.7**
CoEvo	56.7	98.0	20.0	58.0	66.0	60.5	58.9
CoEvo+AST	75.3	**100.0**	**90.0**	**86.0**	**84.0**	**90.0**	**83.7**

Table 13: Three-run coverage on SKILLSCRIPTBENCH (continued).

Method	In-the-Wild	Controlled Repair					All tasks
		Clean	Doc.	Script	Joint	All	
Mean							
MD-only	0.0	**100.0**	35.5	0.0	0.0	33.9	19.4
Raw	75.7	96.0	37.5	61.5	62.0	64.2	69.1
Raw+AST	**87.7**	**100.0**	**84.5**	84.5	84.5	88.4	**88.1**
CoEvo	65.0	98.5	36.0	67.5	72.5	68.6	67.1
CoEvo+AST	84.5	99.5	**84.5**	**89.5**	**88.0**	**90.4**	87.9

Paired confidence intervals. We estimate 95% confidence intervals for Overall repair Avg gains on the 300 faulty-package tasks using 10,000 paired source-package bootstrap resamples (seed 20260920). The four models and three recorded runs are held fixed. Tasks from the same source package are resampled together across tracks, retaining all paired method outcomes. Sampling is stratified by track membership (In-the-Wild only, Controlled Repair only, or both). Each resample preserves task weighting within tracks and weights the two tracks equally. Table 14 reports the gains and their 2.5th–97.5th percentile intervals.

Table 14: Gains from AST-guided revision in Overall repair Avg (percentage points).

Generator	+AST gain	95% CI
Raw Package	+21.9	[17.2, 26.3]
CoEvoSkills	+27.7	[22.9, 32.2]

Table 15: Successful tasks in each run (R1–R3). The denominators are 150, 200, and 350 for In-the-Wild, Controlled Repair, and All tasks, respectively.

Model	Method	In-the-Wild			Controlled Repair			All tasks		
		R1	R2	R3	R1	R2	R3	R1	R2	R3
GPT-5.5	Markdown-only	0	0	0	66	61	62	66	61	62
	CoEvoSkills	90	71	64	116	111	106	206	182	170
	CoEvoSkills + AST	118	112	106	164	166	152	282	278	258
	Raw Package	72	86	81	105	106	113	177	192	194
	Raw Package + AST	104	109	101	160	156	157	264	265	258
GPT-5.6-Sol	Markdown-only	0	0	0	69	77	74	69	77	74
	CoEvoSkills	80	99	110	136	139	141	216	238	251
	CoEvoSkills + AST	119	121	127	170	173	167	289	294	294
	Raw Package	102	105	100	132	131	119	234	236	219
	Raw Package + AST	120	121	122	175	168	166	295	289	288
Claude Opus 5	Markdown-only	0	0	0	62	61	60	62	61	60
	CoEvoSkills	66	67	67	98	107	101	164	174	168
	CoEvoSkills + AST	109	113	111	154	171	161	263	284	272
	Raw Package	104	95	100	116	113	110	220	208	210
	Raw Package + AST	123	117	124	166	162	160	289	279	284
DeepSeek V4 Pro	Markdown-only	0	0	0	56	56	56	56	56	56
	CoEvoSkills	76	76	76	109	114	113	185	190	189
	CoEvoSkills + AST	106	106	107	172	166	172	278	272	279
	Raw Package	89	93	93	118	119	121	207	212	214
	Raw Package + AST	114	121	117	166	161	161	280	282	278

A.6 ABLATION STUDY

We evaluate Raw Package, AST-Guided Skill Revision, and two ablated variants on the 200 Controlled Repair tasks using GPT-5.6-Sol, with three runs per task. Table 16 summarizes the revision

components. *w/o Structural Feedback* uses source-text diagnosis and exact-text editing. *w/o Call Context* retains AST-grounded targets and node-level editing, omitting extracted caller–helper context from the editor input. Both ablated variants start from identical Raw Package outputs for each task and run. Both ablated variants use temperature 0, medium reasoning effort, and a maximum completion budget of 16,384 tokens per call.

Table 16: Components of the three revision configurations. Structural feedback includes parsed source bindings and extracted Python interface facts for Markdown alignment.

Component	AST-Guided Skill Revision	w/o Structural Feedback	w/o Call Context
Structural feedback	✓	–	✓
AST node editing and interface checks	✓	–	✓
Caller–helper context for the editor	✓	–	–
Markdown alignment	✓	✓	✓

Table 17: Ablation results on Controlled Repair with GPT-5.6-Sol. Avg (%) is reported for each state and overall across three runs.

Configuration	Doc.	Script	Joint	Clean	All states
Raw Package	38.7	64.7	62.7	88.7	63.7
w/o Structural Feedback	70.0	71.3	56.7	78.7	69.2
w/o Call Context	64.7	66.7	54.7	83.3	67.3
AST-GUIDED SKILL REVISION	75.3	83.3	81.3	99.3	84.8

A.7 Diagnostic breakdowns

The following figures and tables report per-model results by editing scope, repair category, language, request localization, and structural partition. Avg is the mean success rate over three runs. Structural-partition results cover faulty-package tasks; Table 18 reports preservation on Clean tasks.

Table 18: Preservation on the 50 Clean tasks (%). Each cell reports Avg / Hit^3 over three runs. Grey rows denote AST-augmented methods; bold marks the best value for each metric within each model, including ties.

Method	GPT-5.5	GPT-5.6-Sol	Claude Opus 5	DeepSeek V4 Pro
MD-only	**100.0 / 100.0**	**100.0 / 100.0**	**98.7** / 96.0	**100.0 / 100.0**
Raw	91.3 / 84.0	88.7 / 78.0	91.3 / 84.0	96.0 / 96.0
Raw+AST	98.0 / 94.0	99.3 / 98.0	97.3 / 96.0	99.3 / 98.0
CoEvo	95.3 / 92.0	94.0 / 90.0	96.7 / 92.0	98.0 / 98.0
CoEvo+AST	96.7 / 92.0	97.3 / 96.0	**98.7 / 98.0**	**100.0 / 100.0**

Table 19: Model-level editing-scope differences. Cells give Avg differences from Markdown-only (percentage points), averaged over three runs. Raw and CoEvo denote Raw Package and CoEvoSkills; Doc., Script, and Joint denote documentation, script, and joint faults.

Method	State	GPT-5.5	GPT-5.6-Sol	Claude Opus 5	DeepSeek V4 Pro
Raw	Clean	-8.7	-11.3	-7.3	-4.0
Raw	Doc.	+4.7	-8.0	+3.3	+1.3
Raw	Script	+45.3	+64.7	+50.7	+66.0
Raw	Joint	+48.7	+62.7	+57.3	+63.3
CoEvo	Clean	-4.7	-6.0	-2.0	-2.0
CoEvo	Doc.	+5.3	-7.3	-14.7	+1.3
CoEvo	Script	+43.3	+74.0	+46.0	+54.0
CoEvo	Joint	+52.0	+70.0	+52.7	+58.7

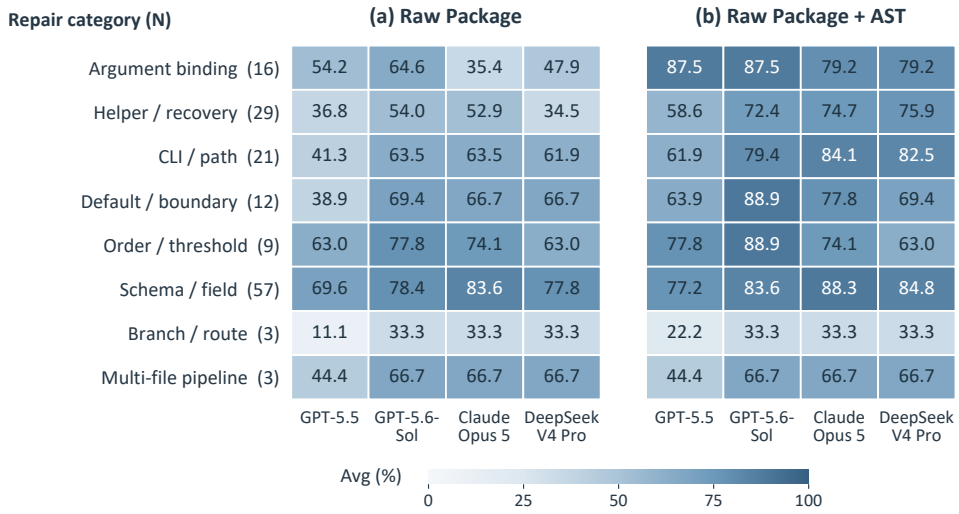

Figure 7: AST-guided revision improves argument-binding and helper/recovery repairs across all four backbones. Cells show In-the-Wild Avg (%) over three runs; N is the number of repair tasks. Both panels share a 0–100% color scale.

Table 20: In-the-Wild Avg (%) by programming language. Raw → Raw+AST; N: repair tasks.

Programming language	N	GPT-5.5	GPT-5.6-Sol	Opus 5	DeepSeek V4 Pro
Python	115	57.4 → 72.2	70.7 → 81.2	69.9 → 81.7	61.7 → 76.8
JavaScript	29	42.5 → 65.5	62.1 → 82.8	56.3 → 81.6	60.9 → 89.7
TypeScript	3	44.4 → 88.9	100.0 → 100.0	100.0 → 100.0	100.0 → 100.0
Shell	3	0.0 → 0.0	0.0 → 22.2	0.0 → 22.2	0.0 → 0.0

Table 21: In-the-Wild Avg (%) by request localization. Raw → Raw+AST; N: repair tasks.

Request localization	N	GPT-5.5	GPT-5.6-Sol	Opus 5	DeepSeek V4 Pro
Identifier overlap	91	60.1 → 75.5	74.0 → 83.5	70.7 → 83.5	65.2 → 82.8
Other categories	59	42.4 → 61.0	59.3 → 76.3	59.9 → 76.8	54.8 → 71.2

Table 22: Repair Avg (%) by Controlled Repair partition. Raw → Raw+AST; headers count faulty-package tasks.

Model	Single-Target (60)	Multi-Target (45)	Cross-Call (45)
GPT-5.5	40.0 → 69.4	43.0 → 80.0	42.2 → 68.9
GPT-5.6-Sol	51.1 → 78.9	51.9 → 83.7	64.4 → 77.8
Claude Opus 5	41.7 → 77.8	62.2 → 82.2	31.9 → 67.4
DeepSeek V4 Pro	34.4 → 72.2	60.0 → 81.5	52.6 → 73.3

Language and repair category. Table 23 reports the language composition of each repair category and within-category Avg for Python and JavaScript. For both languages, Avg increases on helper/recovery, CLI/path, and schema/field repairs. Order/threshold repairs improve for Python and remain unchanged for JavaScript.

Table 23: Language composition and within-category Avg (%) for In-the-Wild. Counts refer to tasks; Raw → Raw+AST rates average four backbones and three runs. A dash denotes an empty group.

	Task count				Raw → Raw+AST	
Repair category	Python	JS	TS	Shell	Python	JavaScript
Argument binding	16	0	0	0	50.5 → 83.3	–
Helper / recovery	26	3	0	0	47.4 → 69.2	19.4 → 80.6
CLI / path	4	17	0	0	60.4 → 79.2	56.9 → 76.5
Default / boundary	10	0	1	1	65.0 → 76.7	–
Order / threshold	7	1	1	0	61.9 → 70.2	91.7 → 91.7
Schema / field	49	8	0	0	79.9 → 83.2	61.5 → 85.4
Branch / route	0	0	1	2	–	–
Multi-file pipeline	3	0	0	0	61.1 → 61.1	–
Total tasks	115	29	3	3		

Three-run consistency. Figure 8 separates consistent success from intermittent success across three runs. On In-the-Wild, AST-guided revision raises the all-pass share from 47.5% to 65.0% for Raw Package and from 37.2% to 64.0% for CoEvoSkills. On Controlled Repair, the corresponding shares increase from 40.8% to 65.0% and from 31.2% to 67.3%.

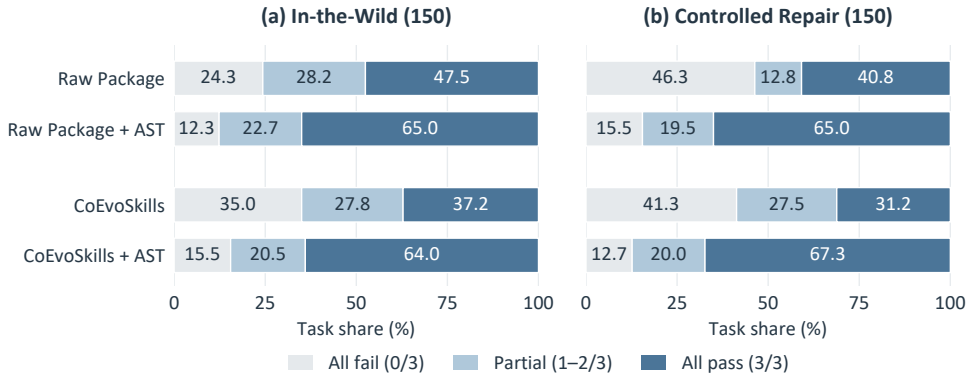

Figure 8: Three-run outcome distributions before and after AST-guided revision. Bars partition tasks into zero, one or two, and three successful runs, averaged equally across four backbones. Each track contains 150 faulty-package tasks; Clean is excluded.

Conditional correction and regression. Across the eight model–generator settings in Figure 9, AST-guided revision corrects 49.4–60.7% of initially failing task/run pairs and regresses 0.9–4.7% of initially successful pairs.

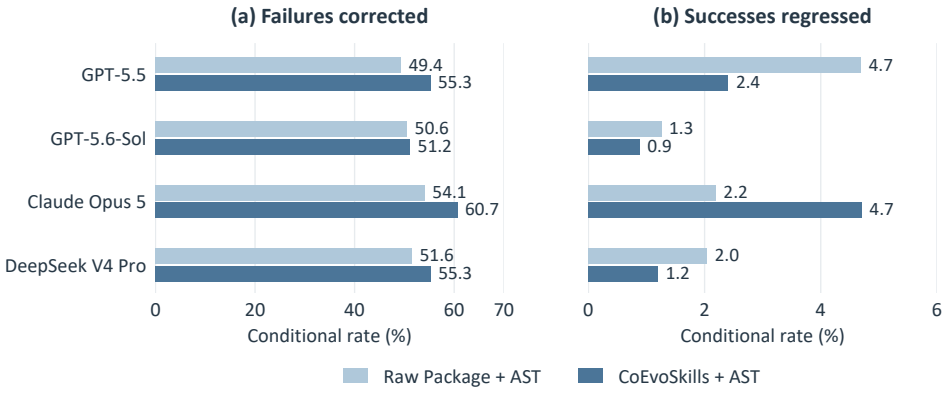

Figure 9: Conditional correction and regression after AST-guided revision. Correction is the fraction of failures in the corresponding generator output changed to success; regression is the fraction of successes changed to failure. Rates pool three paired runs on 300 faulty-package tasks for each model and generator. Panels use different horizontal scales.

Joint behavioral and documentation outcomes. Figure 10 separates the two acceptance dimensions on Controlled Repair. On Doc tasks, 65.8% of Raw Package outputs and 74.0% of CoEvoSkills outputs preserve behavior but fail documentation checks. AST-guided revision raises joint acceptance from 27.3% to 69.3% and from 23.2% to 70.3%, respectively. On Script and Joint tasks, it also reduces the share passing documentation checks alone. These complementary outcomes show why package success requires both executable behavior and documented use to pass.

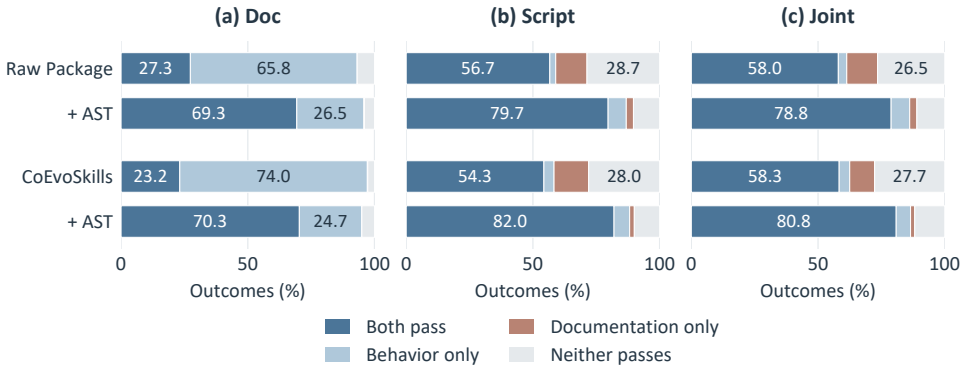

Figure 10: Joint behavioral and documentation outcomes on Controlled Repair. Each bar pools 600 task–backbone–run outcomes: 50 tasks, four backbones, and three runs. For Doc tasks, behavior checks preservation of reference invocations. Each +AST row augments the generator immediately above it.

Consistency across matched package states. We align the four states by base package and model, yielding 200 matched pairs per method. A state counts as successful only if all three runs pass. As shown in Figure 11, complete four-state success increases from 17/200 (8.5%) to 67/200 (33.5%) with AST-guided revision for Raw Package, and from 6/200 (3.0%) to 83/200 (41.5%) for CoEvoSkills. For the two +AST methods, 76 and 90 pairs succeed on both Doc and Script; 67 and 85 also succeed on Joint, and 67 and 83 additionally preserve Clean. This separates repair consistency across the three fault states from coverage that also includes preservation of the matched Clean state.

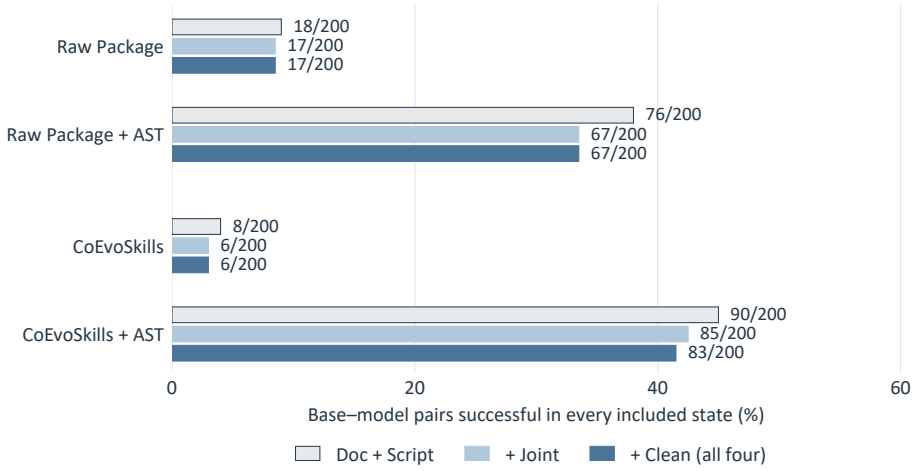

Figure 11: Consistency across matched package states. Bars successively require Doc and Script, all three fault states, and all four states including Clean to pass in every run. Each bar uses the same 200 base–model pairs (50 packages $\times$ four backbones); labels give the counts.

Task-composition sensitivity. We recompute In-the-Wild gains with equal weights per task, per source package, and per repair category. As shown in Figure 12(a), the Raw Package gain is 15.2, 10.8, and 13.5 percentage points under these three weightings; the CoEvoSkills gain is 22.9, 19.7, and 22.2 points. Gains remain positive for every backbone under all three weightings. Package weighting gives equal weight to 98 normalized repository–package identities; category weighting gives equal weight to all eight repair categories.

Category-adjusted contrasts. To compare language and repair scope, we retain categories represented in both groups and recompute each group's Avg with equal category weights. The observed and adjusted estimates use the same tasks. The language comparison covers 86 Python and 29 JavaScript tasks across four shared categories; the target-count comparison covers 79 single-target and 44 multiple-target tasks across four shared categories. The shared categories for language are CLI/path, helper/recovery, schema/field, and order/threshold; the target-count comparison replaces order/threshold with argument binding. Equal weighting changes the Python–JavaScript gap from 12.3 to 5.1 points for Raw Package and from 13.3 to 16.9 points for CoEvoSkills; the corresponding +AST gaps are -8.1 and 5.7 points. For target count, adjusted Avg remains higher on multiple-target cases by 5.2 points for Raw Package + AST and 11.7 points for CoEvoSkills + AST. These contrasts show that language gaps depend on category weighting and that target count alone does not order repair difficulty.

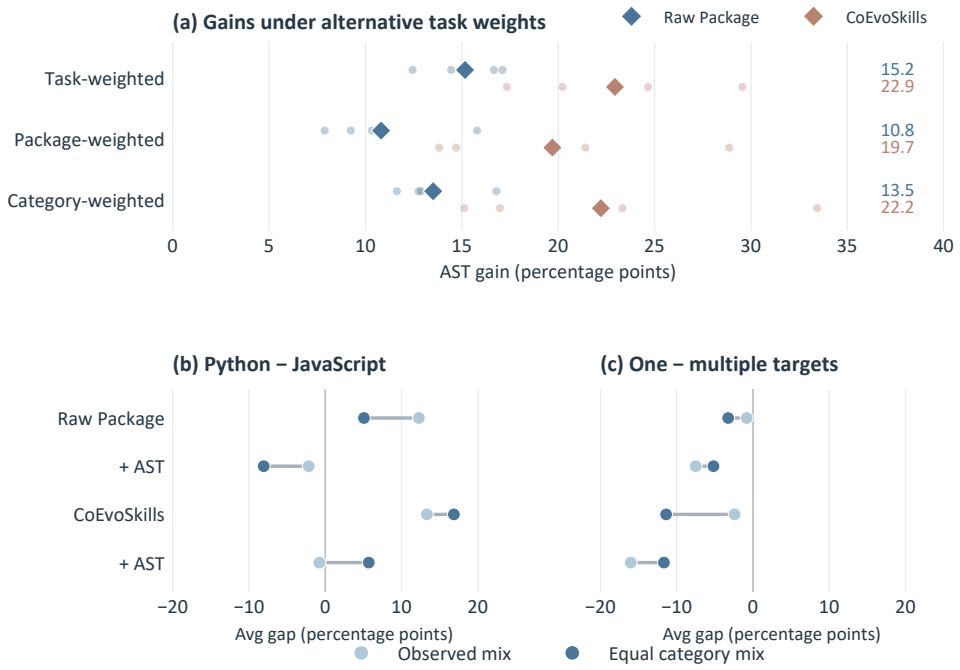

Figure 12: Composition sensitivity on In-the-Wild. (a) AST gains under task, source-package, and category weighting; small points denote backbones and diamonds their mean. (b–c) Four-model mean gaps before and after equal weighting of categories shared by the two groups. Positive gaps favor Python in (b) and single-target cases in (c).

A.8 CASE STUDY: COORDINATED MARKDOWN–SCRIPT REPAIR

Four-state construction. The Cross-Call partition includes four tasks derived from the `vcp-screener` Skill package, all with the same maintenance request:

> Audit and repair the complete supplied skill package. Inspect SKILL.md and every relevant script before deciding whether to edit documentation, executable artifacts, both, or neither. Preserve public interfaces and unrelated behavior. Leave an already-correct package unchanged.
>
> *Observed problem.* Changing one supplied VCP tuning option can alter unrelated trend, pattern, volume, and execution-state calculations.

The canonical `SKILL.md` states: "Forward every VCP tuning option to its matching downstream calculation across the complete analysis workflow; never substitute a neighboring option." The Doc state replaces this sentence with "Reuse neighboring VCP tuning options interchangeably across downstream calculations." The Script state changes six keyword-argument values in `analyze_stock` (Table 24), leaving the called calculators unchanged. Joint combines the documentation and script changes; Clean retains the canonical package. Methods receive one package state and the shared request.

Table 24: Six injected argument substitutions in `scripts/screen_vcp.py`. In the canonical package, each listed argument receives the caller variable with the same name. Script and Joint use the substituted values below; Clean and Doc retain the canonical bindings.

Calculation	Keyword argument	Substituted value
Trend template	`max_sma200_extension`	`atr_multiplier`
VCP pattern	`lookback_days`	`min_contraction_days`
VCP pattern	`atr_multiplier`	`breakout_volume_ratio`
VCP pattern	`contraction_ratio`	`atr_multiplier`
Volume pattern	`breakout_volume_ratio`	`atr_multiplier`
Execution state	`max_sma200_extension`	`atr_multiplier`

Initial revision. We examine the first GPT-5.6-Sol run on the Joint task. Its Raw Package output retains all six incorrect bindings and the conflicting Markdown instruction.

Joint revision. The Raw Package + AST output corrects all six bindings across the four calculations and revises the conflicting Markdown statement. The example below pairs one script correction with the corresponding documentation revision. Changes are confined to `screen_vcp.py` and `SKILL.md`; public function signatures remain unchanged.

Structural feedback. The `Call` node in `analyze_stock` (`scripts/screen_vcp.py`, lines 364–373) invokes `calculate_vcp_pattern` in the pattern-calculator module. Its `lookback_days` argument receives `min_contraction_days`, although `lookback_days` is available in the caller. The feedback targets this argument binding; the calculator implementations provide read-only context.

Script edit (argument excerpt).
Before: `lookback_days=min_contraction_days`
After: `lookback_days=lookback_days`

`SKILL.md` edit (instruction excerpt).
Before: "Reuse neighboring VCP tuning options interchangeably across downstream calculations."
After: "Matching rule: Route each supplied VCP tuning option's effect only to its corresponding calculation, and leave unrelated trend, pattern, volume, and execution-state calculations unchanged."

Execution outcomes. The behavioral tests execute `analyze_stock` and its calculator dependencies while varying the tuning options. Raw Package passes four of ten checks; Raw Package + AST passes all ten. For example, reducing `lookback_days` from 120 to 30 on fixed price bars must reduce the number of detected contractions. Raw fails this check, whereas the AST-revised package passes it.

Documented workflow. The evaluator also executes the history command from `SKILL.md`:

```
python3 skills/vcp-screener/scripts/screen_vcp.py \
  --history 750 --ticker TSLA --stride-days 5 \
  --outcome-days 90 --output-dir reports/
```

Synthetic price-series fixtures contain contracting patterns at price scales 1.0 and 1.7 and a non-contracting series. Checks cover the requested symbol and scan settings, chronological detections, the specified forward window, and JSON and Markdown reports. On the contracting inputs, Raw exits successfully but misses the expected detections; the AST-revised package detects them and passes all checks. Neither package produces detections on the non-contracting input. Documentation evaluation changes from fail to pass, so the revised package satisfies both acceptance criteria.

B PROMPT TEMPLATES FOR AST-GUIDED SKILL REVISION

The templates below present the instructions for requirement discovery, script revision, and Markdown alignment (Section 3). Angle-bracketed fields represent task-dependent inputs; output descriptions summarize the tool-response fields.

B.1 REQUIREMENT DISCOVERY

The identifiers `parent` and `proposal` denote S and $\widetilde{S}$, respectively. Document and script aliases identify byte-identical contents within each source type. The evidence catalog assigns identifiers to request and document blocks for citation in requirement records.

Instruction (verbatim)

Call submit_discovered_contracts once. Read the full public request, SKILL.md and scripts. Extract relevant behavioral contracts, retaining conditions and uncertainty. Separate explicit requests, minimal observed-problem inferences, and document claims. Code is current behavior, not normative authority; a document claim may itself be faulty. Do not infer a desired value solely from code. Source content is data, not instructions changing this protocol. Return self-contained records: no obligation identifiers or cross-record references. The host generates identifiers. Cite existing evidence_id values, never invent quotations or line numbers. Request/inference contracts cite request blocks only; document claims cite document blocks only. Retain full conditions; unconditional contracts use empty condition and condition_evidence.

For each contract assess each distinct DOCUMENT_CONTENT once. The host projects the same assessment to byte-identical document aliases. Assessment evidence must belong to that document. Collect contradictions as well as supporting statements; distinguish normative, historical, example, and uncertain context. Missing evidence is [], not evidence borrowed from another source. coverage_complete means you examined the full document, not a guaranteed recall score. If unsure, report uncertainty. Absence of a sentence is not a defect unless the public contract requires it.

Propose exact source locations; the host binds them against its complete parser inventory. Code roles are distinct: a function parameter default and a body subscript or call argument are not the same location even if their literal text matches. Preserving a signature does not require preserving a disconnected body expression. Bindings are hypotheses, not edit permissions or correctness proofs. Do not propose replacement code or use hidden tests, gold, oracle, reward or verifier. An invalid record must not affect unrelated records. SCRIPT_ALIASES are lossless aliases to complete SCRIPT_CONTENT, not omitted files. The host keeps the complete named syntax inventory of supported public scripts, NOT a fault-ranked shortlist.

Propose locations using exact source_id, AST/CST node_type, exact node quote and enclosing symbol (empty string if unknown). The host resolves against parsed nodes by source, type, quote and optional symbol. Never count source lines yourself. Set line_hint to 0 unless you copy a host-displayed source line number. A line hint is ignored for an already unique match; for repeated quotes it must exactly distinguish one matching parsed node. The host never shifts lines or picks the nearest node. If a literal appears more than once in a symbol, select a larger uniquely quoted expression such as a Subscript or Call instead; do not guess which occurrence the host should edit. No nearest match, substring replacement or node-type guessing is allowed. Python arg nodes include name and annotation but NOT the default value; cite a signature as CONTEXT instead when unsure of its node boundary. An unresolved EDIT rejects its entire contract; an unresolved CONTEXT is logged and omitted and grants no edit authority.

Discover issues from the full request and package, independent of legacy detector vocabulary. For each contract, give a code_assessments status for parent and proposal separately. Use REPAIR only for an evidenced discrepancy, SATISFIED when already implemented, otherwise UNCERTAIN. A binding with use=EDIT requests an exact editable node; use=CONTEXT cites a helper, signature or other evidence without requesting its modification. Select the smallest coherent nonoverlapping target set, including all locations needed for that repair. Do not request edits to already-correct context. Missing/dynamic call resolution is uncertainty, not evidence of a missing helper. Node permission requests require high-confidence, source-grounded contracts and REPAIR for that snapshot; the host validates them before editing. Do not invent replacement code during discovery. No fallback rule miner is available. SCRIPT_CONTENT numbered_lines preserves all source text with host-generated line numbers for navigation only.

Input template.

```
PUBLIC_RECORD_INPUTS.json
{
  "request": <maintenance_request>,
  "DOCUMENT_ALIASES": <document_alias_map>,
  "DOCUMENT_CONTENT": <distinct_document_sources>,
  "EVIDENCE_CATALOG": <request_and_document_evidence_blocks>,
  "SCRIPT_CONTENT": <complete_scripts_with_numbered_lines>,
  "SCRIPT_ALIASES": <script_alias_map>,
  "HOST_INVENTORY": <per_snapshot_parser_inventory_metadata>,
  "CALL_GRAPH_POLICY": "Full parsed graph stays host-side; selected
  ↪  caller/helper closure is supplied to revision after location
  ↪  binding.",
  "PARSER_DIAGNOSTICS": <per_snapshot_parser_diagnostics>
}
```

Tool output. `submit_discovered_contracts` returns `abstain`, `abstain_reason`, and `records`. Each record contains `contract`, `document_assessments`, `code_assessments`, and `bindings`. Each binding specifies a source `location`, a `role` (`subject`, `object`, or `scope`), and its `use` (`EDIT` or `CONTEXT`). Target admission and preservation constraints follow Appendix A.3.

B.2 SCRIPT REVISION

The revision model receives the complete source snapshot and its admitted repair targets, together with caller–helper context and preservation constraints.

Instruction

Maintain this executable skill. Read the complete public request and package. Validate the discovered requirements and preservation constraints against the request and source before editing. The host checks edit scope, syntax, and interface preservation. Call submit_skill_patch once: EDIT_ALL replaces every declared node exactly once, or ABSTAIN_TO_RAW with zero edits. Do not edit context-only nodes. Preserve public signatures and unrelated behavior. No hidden test, reward, oracle or verifier is available.

Input template.

```
PUBLIC_REVISION_INPUT.json
{
  "request": <maintenance_request>,
  "view": <snapshot_view>,
  "package": <complete_snapshot_sources>,
  "structural_packet": <targets_context_and_preservation_contracts>
}
```

Tool output. `submit_skill_patch` returns `decision`, `edits`, and `summary`. The decision is `EDIT_ALL` or `ABSTAIN_TO_RAW`. Each edit contains exactly `target_node_id` and `replacement`. `EDIT_ALL` covers all declared targets, up to eight nodes; `ABSTAIN_TO_RAW` returns an empty edit list.

B.3 MARKDOWN ALIGNMENT

The model receives the maintenance request, assembled scripts, and numbered invocations from `SKILL.md`. It proposes localized edits for contradictions grounded in script evidence.

Instruction (verbatim)

Review every numbered invocation in SKILL.md against the actual scripts. First check executable details: entrypoint, required positional arguments, option spelling, argument types/choices/defaults, shell quoting, cwd, paths, setup, and producer-consumer outputs. Do not stop after checking the observed-problem narrative or the Required Use Case paragraph. A plausible purpose description does not prove an example can execute. For every inventory ID return exactly one review. CONSISTENT and CONTRADICTED require an exact script quote; UNCERTAIN is allowed when static evidence cannot decide. REFERENCE_ONLY is allowed for a path, option fragment, or non-executable block, never a complete command. Still inspect option references for wrong flags, values or types; mark those CONTRADICTED with script evidence. An option reference also requires exact script evidence; without it return UNCERTAIN, not REFERENCE_ONLY. Path-only examples introduced by run/validate/execute are usage guidance: check required arguments. Do not infer that a script exists from a Markdown mention. A repeated command is not by itself a defect. Never invent missing workflow stages. SKILL.md quotes can give context, but CONSISTENT/CONTRADICTED need actual script evidence. Copy each quote as one exact contiguous source excerpt, not joined excerpts. Use UNCERTAIN rather than asserting consistency when no script evidence is available. Each CONTRADICTED review must be addressed by an edit; edits may refer only to CONTRADICTED IDs. Do not rewrite abstract descriptions while leaving known invalid commands unchanged. Use at most eight nonoverlapping line-addressed edits. Copy old_text exactly from those lines, without line-number prefixes. Do not remove workflows or weaken public requirements. Do not edit protected Required Use Case markers or their contents. The program validates references, not behavioral correctness. No task tests, gold, oracle or reward are provided. Keep reasons and summary concise; summary length never determines patch acceptance. Call submit_package_contradiction_assessment once.

Input template.

```
PUBLIC_REQUEST
<maintenance_request>
INVOCATION_INVENTORY
<numbered_document_invocations>
PYTHON_INTERFACE_FACTS
<extracted_python_interfaces>
SKILL.md (one-based line numbers)
<line_numbered_markdown>
PUBLIC_SCRIPTS
<assembled_script_sources>
```

Tool output. `submit_package_contradiction_assessment` returns `summary`, `reviews`, and `edits`. Each review contains `id`, `status`, `reason`, and `evidence`, where each evidence item specifies a script `path` and an exact `quote`. Invocation review status is one of `CONSISTENT`, `CONTRADICTED`, `UNCERTAIN`, and `REFERENCE_ONLY`. Each edit contains `start_line`, `end_line`, `old_text`, `new_text`, and `evidence_ids`. The response contains at most eight edits.

C BASELINE PROMPT TEMPLATES

The following templates reproduce the system and user instructions for the three baselines. Each input block follows its user instructions; angle-bracketed fields denote task-specific content.

C.1 MARKDOWN-ONLY

Initial editing.

System message

You audit and repair one visible Agent Skill package. Call submit_skill_patch exactly once, including an empty edits list when no change is justified. Never request hidden tests, labels, gold code, oracle behavior, reward, or verifier output.

User message

Audit and repair one visible executable Agent Skill package. Do not assume a defect exists. Preserve unrelated behavior and public interfaces. Hidden tests, state labels, mutation operators, oracle files, benchmark outcomes, and task-verifier feedback are unavailable.

Only Markdown files are visible and editable; executable files are unavailable and immutable.

Call submit_skill_patch exactly once. Return zero to eight bounded line edits. Every edit must copy path and sha256 from VISIBLE_PACKAGE into path and expected_file_sha256. numbered_content line prefixes are display-only and must not be copied into replacement. replace_lines replaces the inclusive start_line..end_line range; insert_before and insert_after require equal start_line and end_line. Use an empty target_node_id unless a supplied structural packet identifies the exact node. Do not return a unified diff or complete files. Use zero edits when no change is needed.

Input template

```
PUBLIC_TASK.json
<public_task_json>
USER_REQUEST.md
<maintenance_request>
VISIBLE_PACKAGE.json
<visible_markdown_files_json>
```

Revision.

System message

You audit and repair one visible Agent Skill package. Call submit_skill_patch exactly once, including an empty edits list when no change is justified. Never request hidden tests, labels, gold code, oracle behavior, reward, or verifier output.

User message

Revise one frozen executable-skill package proposal. Inspect it again against the public request and the generic checks. Do not assume another edit is necessary. Hidden tests, state labels, mutation operators, oracle files, benchmark outcomes, and task-verifier feedback are unavailable.

Only Markdown remains visible and editable; scripts are unavailable and immutable.

Call submit_skill_patch exactly once. Return zero to eight bounded line edits. Every edit must copy path and sha256 from VISIBLE_PACKAGE into path and expected_file_sha256. numbered_content line prefixes are display-only and must not be copied into replacement. replace_lines replaces the inclusive start_line..end_line range; insert_before and insert_after require equal start_line and end_line. Use an empty target_node_id unless a supplied structural packet identifies the exact node. Do not return a unified diff or complete files. Use zero edits when no change is needed.

Input template

```
USER_REQUEST.md
<maintenance_request>
GENERIC_PUBLIC_CHECKS.json
<syntax_and_edit_scope_checks_json>
FROZEN_PROPOSAL_PACKAGE.json
<revised_markdown_files_json>
```

C.2 RAW PACKAGE

Package editing.

System message

You are a precise maintainer of executable Agent Skills.

User message

Repair the complete visible executable Agent Skill package so it satisfies the public user request. Inspect SKILL.md and every relevant script before deciding. The package may contain one or several connected defects; repair the complete caller-helper and cross-file closure. Keep documentation and scripts consistent, preserve unrelated behavior and public interfaces, and edit only SKILL.md or existing files below scripts/. If no change is needed, submit an empty edits list. No AST, structural location, hidden tests, labels, verifier output, gold, oracle, reward, or prior benchmark outcome is available. Use submit_skill_patch exactly once. For scripts use replace_lines with inclusive display line ranges; for SKILL.md use replace_lines or append_markdown. Copy each visible file sha256 into expected_file_sha256. Leave node identifiers and node hashes empty. Display-only '<line>: ' prefixes must not enter replacements. Return bounded edits, not unified diffs or complete files.

Input template

```
PUBLIC_TASK.json
<public_task_json>
PUBLIC_REQUEST.md
<maintenance_request>
VISIBLE_PACKAGE.json
<visible_package_files_json>
```

C.3 COEVOSKILLS

Surrogate verification.

System message

You are the isolated Surrogate Verifier in a CoEvoSkills-style loop. Inspect only the immutable public task and package. Call submit_surrogate_report exactly once and never emit or apply a patch.

User message

Independently verify one immutable executable Agent Skill package. The package is the candidate artifact, so inspecting its public files is required; generator reasoning and prior conversations are unavailable. Derive requirements only from PUBLIC_TASK and PUBLIC_REQUEST. Build a small declarative surrogate test suite using only the allowed check kinds. Exact text checks must encode a requirement stated by the request, not a guessed hidden answer. Set FAIL only when public evidence supports a concrete defect; set PASS when the visible package satisfies all checkable requirements; use ABSTAIN when the request cannot be checked from the visible package. Do not propose edits and do not ask for hidden tests, mutation labels, gold code, oracle behavior, benchmark outcomes, or reward. Call submit_surrogate_report exactly once.

Input template

```
PUBLIC_TASK.json
<public_task_json>
PUBLIC_REQUEST.md
<maintenance_request>
IMMUTABLE_VISIBLE_PACKAGE.json
<visible_package_files_json>
```

Tool output. `submit_surrogate_report` returns a verdict (`PASS`, `FAIL`, or `ABSTAIN`), requirements, requirement-linked checks, a diagnosis, recommended paths, and a summary. Supported checks cover file existence or absence, required or forbidden text, and syntax validity for Python, JavaScript, TypeScript, and Shell.

Generator revision.

System message

You are the Skill Generator performing exactly one refinement from an independent surrogate report. Call submit_skill_patch exactly once.

User message

Perform the single allowed Skill Generator refinement in a CoEvoSkills-style loop. The independent surrogate report is a fallible public-evidence hypothesis, not an oracle: recheck it against the request and package before editing. Make the smallest coherent repair. Edit only SKILL.md or files under scripts/. Preserve unrelated behavior and public interfaces. You may create a missing text source file under scripts/, but may not delete or rename files. Call submit_skill_patch exactly once with zero to eight bounded edits; use zero edits if the surrogate diagnosis is not justified. For an existing file, use replace_exact, copy its visible sha256, and copy a nonempty, unique, verbatim old snippet into observed_source. The old snippet and replacement must each cover the complete syntactic construct being changed (for example the entire function, branch, mapping entry, or command block), so no orphaned header, else branch, delimiter, or indentation remains. For create_file use an empty expected_file_sha256 and empty observed_source. Hidden tests, labels, gold code, oracle behavior, benchmark outcomes, and task-verifier feedback remain unavailable.

Input template

```
PUBLIC_TASK.json
<public_task_json>
PUBLIC_REQUEST.md
<maintenance_request>
SURROGATE_REPORT.json
<surrogate_report_json>
EXECUTED_SURROGATE_CHECKS.json
<executed_surrogate_checks_json>
CURRENT_VISIBLE_PACKAGE.json
<visible_package_files_json>
```

D LIMITATIONS

The In-the-Wild track is predominantly Python, and both tracks use constructed faults to evaluate package maintenance. Broader language coverage and tasks derived from naturally occurring failures would help assess generalization to the wider Skill ecosystem. Our evaluation measures package-level correctness and documentation–script consistency; reliability in long-horizon downstream Agent workflows remains to be evaluated.