SWE-SKILLS-BENCH: DO AGENT SKILLS ACTUALLY HELP IN REAL-WORLD SOFTWARE ENGINEERING?

Anonymous authors
Paper under double-blind review

ABSTRACT

Agent skills, structured procedural knowledge packages injected at inference time, are increasingly used to augment LLM agents on software engineering tasks. However, their real utility in end-to-end development settings remains unclear, and no standard procedure tests an individual skill before installation. We present **SWE-Skills-Bench**, a skill-centric, requirement-driven benchmark that measures the extent to which individual agent skills improve performance on real-world software engineering (SWE) tasks. We start from 98 public SWE skills, each paired with a real GitHub repository pinned at a fixed commit. For each skill, our pipeline (i) reverse-engineers ten requirements specific to that repository, (ii) turns each requirement's acceptance criteria into deterministic, execution-based verifiers, and (iii) builds a reproducible container environment. Under each of four agent–model configurations, we run every task once without the skill and once with it, yielding 3,920 paired evaluations. For each requirement, we compute a paired delta, i.e., the change in outcome when the skill is added. We then aggregate these deltas across a skill's ten requirements into a per-skill utility profile, and use a statistical test to classify each skill as Golden (significantly helpful), Ineffective (no significant effect), or Harmful (significantly detrimental). Our results show that skills provide limited benefits: 92 of 98 skills have no statistically significant effect, and adding skills raises the average pass ratio only from 76.6% to 76.9%. The few helpful skills supply procedures that the agent would otherwise omit (gains of up to $+17.7$ points). In contrast, harmful skills anchor the agent on their own templates and override the task's explicit instructions (losses of up to -17.0 points). Moreover, a skill's utility depends on the agent harness and model: 40 of 98 skills flip the sign of their effect between two harnesses running the same model. These findings suggest that skill utility is an empirical, configuration-dependent property rather than an intrinsic one, and SWE-Skills-Bench provides a testbed for testing skills before deployment. Our code is available at https://anonymous.4open.science/r/SWE-Skills-Bench.

1 INTRODUCTION

LLM-based agents (Yao et al., 2023; Schick et al., 2023) have been increasingly deployed across multiple software engineering (SWE) tasks, from bug fixing (Jimenez et al., 2024; Zhang et al., 2024; Wang et al., 2025; Yang et al., 2025) to infrastructure management (Yang et al., 2024; Song et al., 2025). Agent skills are structured markdown packages that encode procedural knowledge, code templates, and domain conventions for consumption by LLM-based agents (Anthropic, 2025b; Fang et al., 2025; Wang et al., 2024a;b; Xu & Yan, 2026; Wu & Zhang, 2026). At inference time, a skill is injected into the agent's context window as a reference document. Unlike fine-tuning or retrieval-augmented generation, skills require neither model modification nor an external retrieval pipeline. Figure 1 illustrates how an agent uses skills to solve a software engineering task. Public skill ecosystems have grown rapidly. Skills.sh, a popular skill website, tracked over 90K installable skills in early 2026 (Virtual Uncle, 2026) and more than 1.55M as of September 2026 (Vercel, 2026).

This rapid proliferation has outpaced quality assurance. A public skill can be installed and reused immediately, yet there is no standard way to test whether it actually helps. Simply observing whether an agent completes a task is not enough, because the outcome mixes together the capability of the foundation model, the behavior of the agent harness, the difficulty of the task, and the contribution of

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.

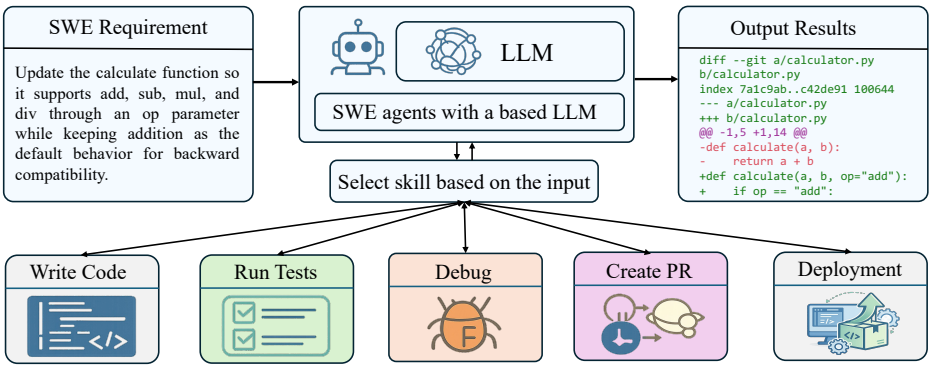

Figure 1: Illustration of how agent skills are used in a software engineering workflow. Given a natural-language requirement, the LLM-based agent selects the most relevant skill from its skill library, including skills such as writing code, running tests, debugging, creating pull requests, and deploying, and injects it into the context window. The agent then executes a series of SWE actions to produce the final software artifacts (such as code) that fulfill the requirement.

the skill. Isolating the skill's contribution requires comparing the same task with and without the skill. Existing benchmarks support such skill-level comparison only partially. Terminal-Bench (Merrill et al., 2026) evaluates realistic command-line tasks, HumanEval (Chen et al., 2021), DS-1000 (Lai et al., 2022), BigCodeBench (Zhuo et al., 2025), and LiveCodeBench (Jain et al., 2025) evaluate code generation, and agent benchmarks such as AgentBench, τ-Bench, AppWorld, and MLE-Bench (Liu et al., 2024; Yao et al., 2025; Trivedi et al., 2024; Chan et al., 2025) evaluate tool use and end-to-end task completion. None of these treats a skill as the intervention under test. The closest work is SkillsBench (Li et al., 2026), which evaluates agents with and without skills on 87 expert-contributed tasks across eight domains, using containerized environments and deterministic verifiers. However, SkillsBench is *task-centric* but not *skill-centric*: each evaluation starts from a fixed, expert-contributed task and asks whether providing curated skills for that task improves agent performance. Furthermore, SkillsBench is not specifically designed for software engineering: software engineering is only one of its eight domains, and its primary goal is to measure broad cross-domain skill efficacy rather than requirement satisfaction in real-world development workflows.

A principled benchmark for SWE skill utility must answer a deceptively simple question: *Does the skill help the agent satisfy the requirements it is intended to support?* Software engineering is inherently requirement-driven (Sommerville, 2015; Zave, 1997; Pohl, 2010; Nuseibeh & Easterbrook, 2000; Boehm, 1981): a task succeeds when every acceptance criterion stated in its specification is met, and unit tests serve as executable encodings of those criteria. We therefore reverse-engineer repository-specific requirements from each skill's claimed procedures, then systematically translate their acceptance criteria into deterministic verifiers. Building on this principle, we present **SWE-Skills-Bench**, a skill-centric framework for constructing and conducting automated evaluations of software engineering skills. We manually curate 98 public SWE skills and pair each with an authentic GitHub repository pinned at a fixed commit. For every skill–repository pair, our pipeline (i) reverse-engineers ten skill-triggering requirements, (ii) synthesizes requirement-driven executable verifiers, (iii) constructs a reproducible Docker environment for the target repository, and (iv) conducts matched skill-free and skill-assisted executions across agent–model configurations. Because the paired arms differ only in whether the skill is available, the resulting change in pass ratio can be attributed to the tested skill under that configuration.

Our main contributions are as follows:

- **Automated skill-centric evaluation.** We formulate skill evaluation as an executable, requirement-driven procedure. We then build an end-to-end pipeline that takes a curated skill–repository pair as input, generates requirements, deterministic verifiers, and a containerized environment, and outputs a utility profile for the skill.
- **Empirical findings.** ① Skills do not provide a universal performance boost: 92 of 98 skills are Ineffective, and the mean pass ratio changes only from 76.6% without skills to 76.9% with skills (+0.3 points). ② Four skills are Golden, and they help by supplying procedures the agent would otherwise omit. For example, `viper-configuration` (+17.7 points) leads the agent to

use the `spf13/viper` library that the requirements expect. Without the skill, half of the runs never add the library and instead write their own configuration loader. ③ Skills can also hurt: `asyncio-patterns` lowers the pass ratio by 17.0 points on average, and trajectories show a skill's concrete templates overriding the task's explicit instructions, for example by relocating the required files. ④ Skill utility is configuration-dependent. Skill-level deltas are nearly uncorrelated across configurations. 40 of 98 skills change sign between two harnesses running the same model. These results establish that SWE skill utility is highly domain-specific, configuration-dependent, and context-dependent, favoring targeted skill design over blanket adoption.

2 Related Benchmarks & Datasets

We organize related work into two threads: SWE- and Skill-related benchmarks. Generally, SWE-related benchmarks do not include skills in their evaluations, whereas skill-related benchmarks do not focus specifically on software engineering tasks. To the best of our knowledge, we are the first skill-centric benchmark for software engineering. Table 1 summarizes the key differences.

Table 1: Comparison of SWE-Skills-Bench with existing benchmarks. "Skill Cond." indicates whether the benchmark includes agent skills. "Det. Verifier" indicates whether deterministic (non-LLM) verification is included. "SWE-Focused" indicates whether the benchmark is specifically designed for software engineering tasks.

Benchmark	Size	Skill Cond.	Real Projects	Det. Verifier	SWE-Focused
SWE-Bench Verified	500	None	Yes	Yes	Yes
TerminalBench	89	None	Yes	Yes	Yes
HumanEval	164	None	No	Partial	No
SkillsBench	87	Yes	Yes	Yes	Partial
SWE-Skills-Bench	**98**	**Yes**	**Yes**	**Yes**	**Yes**

SWE-related Benchmarks. This line of work can be further divided into repository-level SWE benchmarks and code-generation benchmarks. The former focus on realistic, project-level software engineering tasks with execution-based verification. SWE-Bench Verified (Jimenez et al., 2024) is a human-validated subset of 500 instances from SWE-Bench, drawn from 12 Python repositories and evaluated via fail-to-pass tests. TerminalBench (Merrill et al., 2026) evaluates agents on 89 realistic CLI tasks in containerized environments. However, these benchmarks do not isolate the marginal benefit of injecting procedural skill documents. Code generation benchmarks, in contrast, mainly evaluate models on self-contained coding problems (often algorithmic or snippet-level) without full project context. HumanEval (Chen et al., 2021) comprises 164 hand-crafted programming challenges at the function level, and therefore does not capture multi-file reasoning, dependency management, or end-to-end SWE workflows.

Skills Benchmarks. SkillsBench (Li et al., 2026) takes an important first step toward benchmarking skills as first-class artifacts by comparing agent performance across different skill conditions. Nevertheless, it is not SWE-specific: software engineering forms only a limited subset of its task suite, and the benchmark is not designed around the central success criterion in real-world development—whether explicit requirements are satisfied in repository-grounded workflows. Our work addresses this gap by constructing a requirement-driven benchmark focused exclusively on SWE, pairing each skill with fixed-commit repositories, explicit requirements, and deterministic, execution-based verification.

3 SWE-Skills-Bench: Requirement-Driven Skill Evaluation

3.1 Overview and Problem Formulation

SWE-Skills-Bench evaluates a skill by measuring how much it improves the agent's performance on requirements, which are reverse-engineered from the skill content. The input is a manually curated skill–repository pair (s, R_s), where R_s is an authentic GitHub repository pinned at a fixed commit. Once the skill and its repository are fixed, we proceed in three steps. First, we reverse-engineer repository-specific requirements from the skill's content. Second, we build executable verifiers for

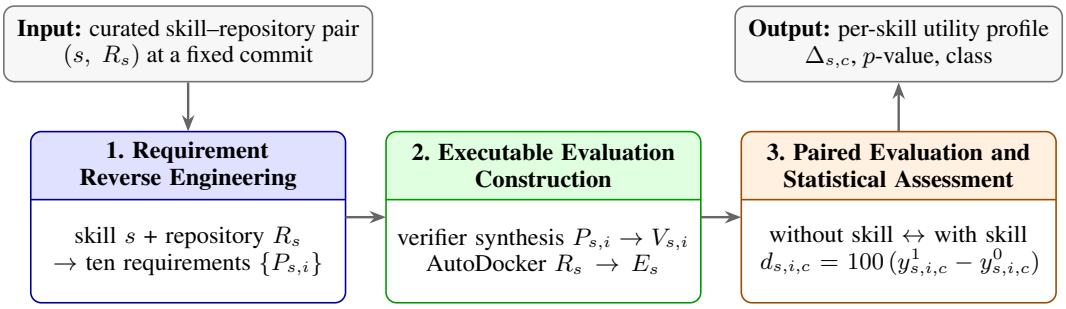

Figure 2: Overview of the SWE-Skills-Bench pipeline. Given a curated skill s and a repository R_s at a fixed commit, SWE-Skills-Bench (1) reverse-engineers ten requirements from the skill, (2) builds a verifier based on each requirement and a reproducible environment, and (3) runs each requirement with and without the skill to produce a per-skill utility profile.

these requirements and a reproducible environment for running them. Third, we run each requirement twice, once with the skill and once without it. Figure 2 showcases the pipeline.

For each skill s, we construct $m = 10$ requirements $\mathcal{P}_s = \{P_{s,1}, \ldots, P_{s,m}\}$, a containerized environment E_s, and a deterministic verifier $V_{s,i}$ for each requirement. Let A_c denote an agent–model configuration c and let $z \in \{0, 1\}$ indicate whether s is unavailable or available, respectively. The agent produces an artifact

$$x^z_{s,i,c} = A_c(R_s, E_s, P_{s,i}; s^z), \qquad s^z = \begin{cases} s, & z = 1, \\ \varnothing, & z = 0, \end{cases} \tag{1}$$

which receives the execution-based pass ratio $y^z_{s,i,c} = V_{s,i}(x^z_{s,i,c}) \in [0, 1]$, the fraction of executable checks the artifact passes. Within a matched pair, the repository, requirement, environment, verifier, and agent–model configuration are held fixed; only access to s changes.

Evaluation inputs. We manually curate a skill subset through a three-stage filtering pipeline as evaluation inputs. First, we scan the `mcpmarket` category leaderboard and select six of the nine core categories that best align with software-engineering workflows and are amenable to unit-test evaluation: Developer Tools, Security & Testing, API Development, Data Science & ML, Deployment & DevOps, and Analytics & Monitoring. Second, we apply semantic filtering to exclude generative or subjective skills, retaining only those that target concrete SWE actions such as *fix*, *build*, and *develop*. Third, we exclude candidates whose associated repositories are prohibitively large or incur high environment and setup costs. This pipeline yields 98 skills distributed across the six categories: API Development (26), Developer Tools (18), Data Science & ML (17), Deployment & DevOps (14), Analytics & Monitoring (12), and Security & Testing (11).

3.2 Requirement Reverse Engineering

We give the skill document, the repository metadata, and the relevant project structure to a fixed requirement-engineer prompt. The prompt produces ten distinct requirements, each designed to exercise a different aspect of the skill. In contrast to task-centric construction, the skill determines the target capability first, and each requirement is then written to test it.

Each i-th requirement of skill s, $P_{s,i}$, follows a standardized template with four fields: (i) *Background*, which supplies repository context; (ii) *Requirement*, which states the intended change; (iii) *File Operations*, which identifies artifacts that may need to be created or modified; and (iv) *Acceptance Criteria*, which specify observable success conditions. Figure 12 shows the prompt used to generate the requirements, while Figure 13 provides an example of a generated requirement.

3.3 Executable Evaluation Construction

Natural-language requirements become executable evaluation instances only after we construct both a scoring procedure and a reproducible environment. We implement these two components as requirement-driven verification and AutoDocker environment construction.

Requirement-driven Verification. Rather than relying on subjective judgments, we convert every acceptance criterion in the requirement document $P_{s,i}$ into objective, deterministic tests, ensuring that each test outcome is directly traceable to a specific requirement. We provide $P_{s,i}$ (together with repository metadata such as repo path, language, and available test commands) to a fixed "professional test engineer" prompt template, which instructs the model to (i) enumerate testable behaviors from each acceptance criterion, (ii) instantiate representative and edge-case scenarios, and (iii) encode them into a deterministic `pytest` test file with strong discriminative power (i.e., tests must run the produced code and verify concrete outputs/structures rather than keyword-level heuristics). The prompt also enforces structural constraints such as a minimum number of test cases and per-test docstrings. The prompt template is shown in Figure 11.

AutoDocker environment construction. Our AutoDocker module provisions E_s from the pinned repository and its environment specification. It selects a language-appropriate base image, restores a clean checkout at the designated commit, and either executes explicit setup commands or detects dependency installation from common project manifests and lock files. The resulting container fixes the workspace, dependencies, CPU and memory limits, and command timeouts used by both conditions. Before evaluation, pre-existing agent-skill directories are removed so that the assisted condition exposes only the skill under test. This construction ensures that environmental variation cannot be mistaken for skill utility.

3.4 Paired Evaluation and Statistical Assessment

For each pair of skill–requirement–configuration $(s, P_{s,i}, c)$, we initialize two copies of the same container state and provide the same requirement to the agent. In the assisted condition, we invoke skill s explicitly at the start of the prompt through the harness's native command (`/skill` in Claude Code, `$skill` in Codex), which loads the full skill document into the agent's context, whereas in the unassisted condition, the skill is absent. The requirement-level delta, expressed in points, and the skill-level delta under configuration c are

$$d_{s,i,c} = 100\left(y^1_{s,i,c} - y^0_{s,i,c}\right), \qquad \Delta_{s,c} = \frac{1}{m}\sum_{i=1}^{m} d_{s,i,c}. \tag{2}$$

Positive $\Delta_{s,c}$ indicates improvement under configuration c, whereas negative values indicate interference; the overall skill-level delta Δ_s averages each requirement's delta across configurations before averaging over requirements.

To decide whether a skill has a reliable effect, we test its ten requirement-level deltas with an exact two-sided paired sign-flip test. Intuitively, the test asks whether the ten deltas favor one direction more consistently than chance alone would produce. The p-value is the probability of seeing a skill-level delta this large if the skill had no real effect. We call the resulting summary of a skill its *utility profile*. For each configuration, it contains the requirement-level deltas, the skill-level delta, the p-value, and the resulting class. A skill is *Golden* if its skill-level delta is positive with $p < 0.05$, *Harmful* if it is negative with $p < 0.05$, and *Ineffective* otherwise. Ineffective means that no directional effect was detected with ten requirements, not that the effect is zero.

4 Results of SWE-Skills-Bench

4.1 Experimental Setup

We evaluate 98 skills, each with ten requirements, under four agent–model configurations: Codex with GPT-5.4-mini or Kimi-K2.6, and Claude Code (Anthropic, 2025a) with Kimi-K2.6 or Haiku 4.5. Each requirement is evaluated with and without the target skill in the containerized environment described in Section 3.4. The evaluation spans 980 distinct requirements, 392 skill configuration profiles (agent–model), and all 3,920 expected skill-free/skill-assisted pairs. Runs affected by infrastructure failures, such as unreachable model endpoints, rate limiting, or lost containers, are re-executed until a clean run is obtained. Each pass ratio is bound to the trajectory of the run that produced it.

4.2 Evaluation Results

For each agent–model configuration, the evaluation assigns every skill a utility profile. Figure 3 shows the requirement-level deltas for all skills, and Table 2 compares average performance with and

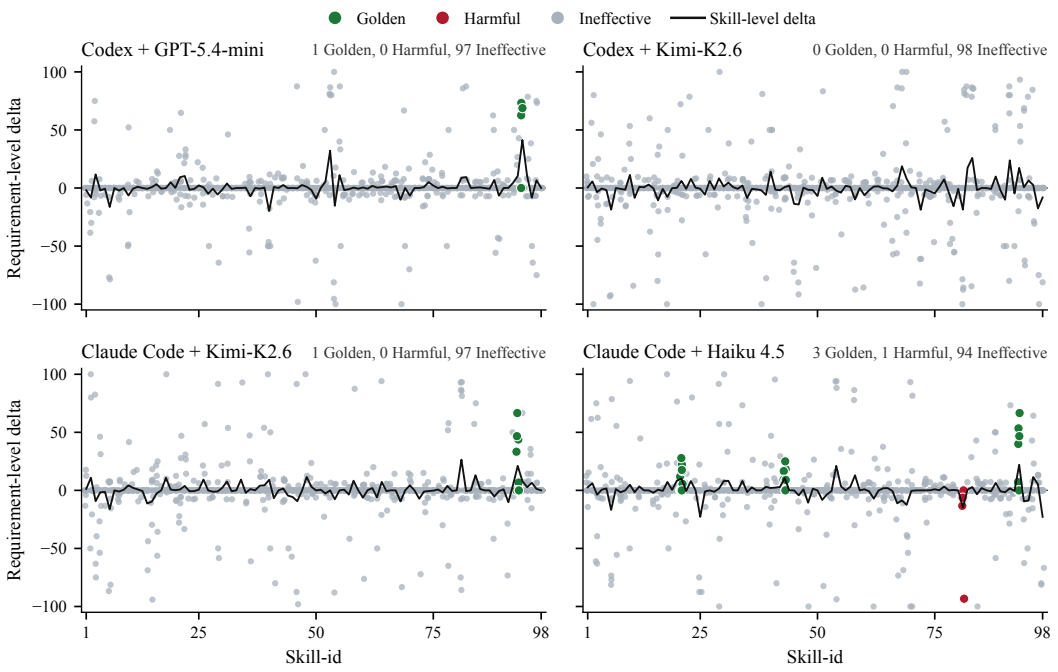

Figure 3: Requirement-level deltas for every skill under each configuration: one point per skill-free/skill-assisted pair (980 per panel), skills in alphabetical order with the same index in all panels. The line connects skill-level deltas; colors give the configuration-specific classification (green Golden, red Harmful, gray Ineffective).

without skills. Skill-level deltas are close to zero. In every configuration, the mean pass ratio changes by less than one point. Requirement-level deltas are occasionally large (± 50 points or more), but gains and losses occur at similar rates and cancel out within most skills.

Finding 1: Most skills show no detectable directional effect. Under our classification protocol, 92 of the 98 skills (93.9%) are classified as Ineffective, compared with four Golden and two Harmful skills. Skill access increases the aggregate pass ratio only from 76.59% to 76.93%, and the difference remains below 1% in every agent–model configuration. As shown in Figure 3, skill-level deltas are therefore small for most skills and directionally consistent for only a few.

At the requirement level, 2,711 of the 3,920 pairs (69.2%) have identical verifier outcomes with and without the skill. In 1,313 pairs, both runs pass every check; in the remaining 1,398, both pass the same subset of checks. These unchanged outcomes are not explained solely by ceiling effects. Among the skills classified as Ineffective, 35 have unassisted pass ratios below 80%, yet for 20 of them, the absolute difference between the assisted and unassisted pass ratios is below 0.02.

The small aggregate effect nevertheless masks occasional large changes. In 240 pairs (6.1%), the pass ratios under the two conditions differ by at least 0.5. In approximately half of these pairs, one run passes fewer than 20% of the checks while the other passes more than 80%. Thus, the near-zero average reflects a combination of many unchanged outcomes and a small number of large gains and losses, rather than uniformly small effects across all requirements.

Finding 2: Only a few skills improve performance overall. Four skills receive an overall Golden classification under our protocol: `viper-configuration`, `go-testify-patterns`, `junit5-testing`, and `fastapi-patterns` (Table 3). Among them, `viper-configuration` produces the largest improvement, increasing its aggregate pass ratio from 59.61% to 77.30%. Its effect is positive in all four agent–model configurations and in nine of the ten requirements.

The `viper-configuration` case in Figure 4 illustrates one way in which a skill can help. Its requirements exercise configuration workflows involving namespaced environment variables, override precedence, typed decoding, and isolation between configuration loads. Across the ten requirements and four configurations, all 40 assisted runs integrate `spf13/viper`, whereas only 20 of the 40

Table 2: Mean pass ratio over paired with/without runs. Each configuration contributes 980 pairs, and the last row pools all 3,920 pairs. Pass ratios are percentages, and Δ is the mean requirement-level delta over the row's pairs. Green and red mark observed direction only.

Agent	Model	Without Skill	With Skill	Δ (points)
Codex	GPT-5.4-mini	75.98	76.74	+0.76
Codex	Kimi-K2.6	77.60	77.32	−0.29
Claude Code	Kimi-K2.6	78.53	79.18	+0.64
Claude Code	Haiku 4.5	74.24	74.50	+0.26
All pairs		**76.59**	**76.93**	**+0.34**

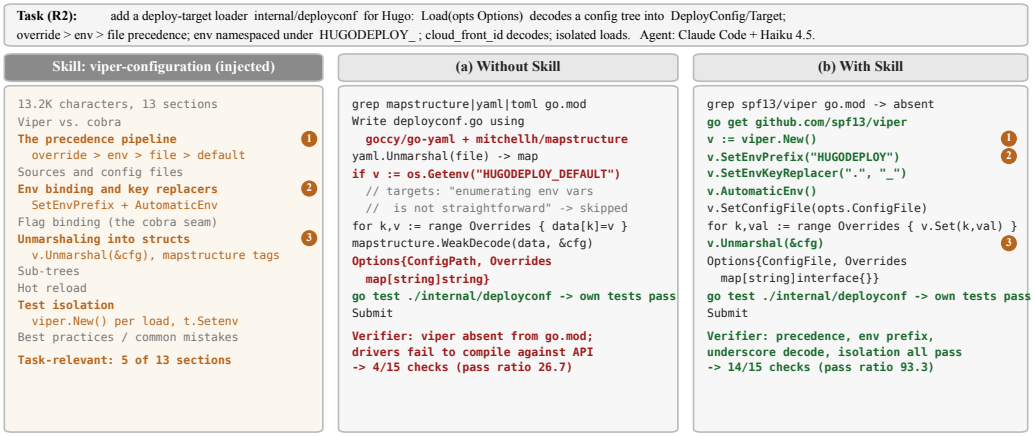

Figure 4: Case study of the `viper-configuration` skill on a Hugo deployment task (R2). The agent must implement a configuration loader that respects a specific precedence order (override > env > file) and handles environment variable namespacing. Without Skill (a), the agent fails to identify the standard library (viper) and instead writes a fragile, hand-rolled YAML parser. This results in a low pass ratio (26.7%) as it misses critical features like automatic env binding. With Skill (b), guided by the skill's procedural knowledge, the agent correctly imports viper and implements the required logic (isolated instance, env prefix, unmarshalling), achieving a high pass ratio (93.3%).

unassisted runs do so. Assisted agents also more consistently follow the corresponding Viper workflow. In contrast, unassisted agents often construct ad hoc configuration loaders that pass their self-authored tests but fail verifier checks involving compilation, precedence, decoding, or isolation. The skill therefore provides a procedural mapping from the required configuration behavior to concrete library operations that the unassisted agent frequently omits.

The remaining Golden skills show smaller gains, and their effects vary across requirements and configurations. The Viper case should therefore be viewed as an illustrative example rather than a general explanation for all positive skill-level deltas.

Finding 3: Skills can actively degrade performance through context interference. Two skills receive a Harmful label overall: `grafana-dashboards` (−2.96 points) loses a few points on seven of its ten requirements, and `asyncio-patterns` (−16.96 points) falls from a pass ratio of 83.20 to 66.24, with negative deltas on eight of ten requirements and in every configuration (−16.12 to −18.61 points). Regressions of this kind also appear within single configurations. `github-actions-templates` is Ineffective overall (−5.14 points) but loses 22.78 points under Claude Code with Haiku 4.5, where three of its ten requirements collapse from $75-100$ to $0-12.5$. These regressions stem from the mismatch between the *holistic* scope of a skill and the *focused* requirements of individual tasks. Each skill is authored as a comprehensive reference for its technical domain, encoding best practices that span project layout and configuration templates. When a task exercises only a narrow slice of this knowledge, the surplus context can interfere with the agent's work. Figure 5 illustrates an example of skill `github-actions-templates`. The task asks the

agent to add four reusable CI/CD workflows to `actions/starter-workflows` at explicitly named paths under `ci/`. The skill packages workflow templates covering testing, Docker build-and-push, Kubernetes deployment, matrix builds, reusable workflows, security scanning, and deployment approvals. Among them, the *Reusable Workflows* section contains a template of exactly the kind of workflow the task requires. But the file location is different. The template's first line places it at `.github/workflows/reusable-test.yml`, the directory in which GitHub executes a repository's own workflows, whereas the task names `ci/`. Because the template matches the task so closely, the agent copies its details, including this location, instead of following the task's instructions.

Without the skill, the agent follows the requirement. It writes the four files under `ci/`, references the composite action as `./composite-setup`, and passes all 16 verifier checks. With the skill, the agent adopts the template's location instead and writes the files under `.github/workflows/ci/`, adjusting even the composite-action reference to the new path. The verifier reads the files from the paths the requirement specifies, finds none of them, and 14 of its 16 checks fail (pass ratio $100 \rightarrow 12.5$). The same displacement recurs on the two other collapsed requirements (R4 and R5), whose files also migrate into `.github/workflows/`, which accounts for the entire loss of this skill under Haiku 4.5. Misplaced deliverables are a recurring signature of harmful injection. Among the 118 requirement pairs whose delta is -50 points or worse, the verifier reports the assisted run's deliverable as missing or misplaced in 39 (33%), never the unassisted run's. This explains the seemingly paradoxical outcome. A skill containing *objectively relevant* content nonetheless *degrades* performance. The practical implication is that skill design should favor abstract guidance patterns over concrete, opinionated templates with hard-coded paths and values, as the latter risk anchoring the agent on specifics that the task overrides.

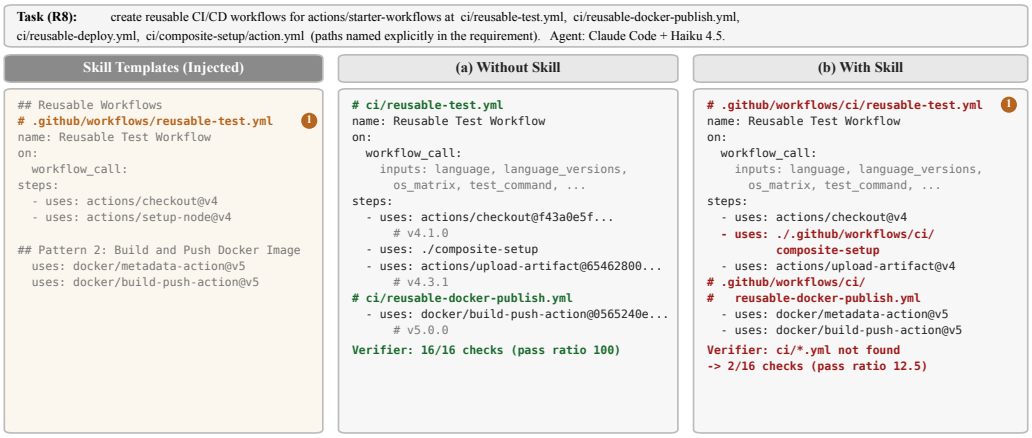

Figure 5: Case study of context interference from the `github-actions-templates` skill (Claude Code with Haiku 4.5, R8, $100 \rightarrow 12.5$). The task requires four reusable workflows at explicitly named paths under `ci/`. **(a)** Without the skill, the agent writes the files where the requirement specifies and passes all 16 checks. **(b)** With the skill, the agent adopts the location named in the skill's template and writes the files under `.github/workflows/ci/`.

Finding 4: Skill utility depends on the agent–model configuration. Skill-level deltas transfer poorly across configurations. Only six skills are positive, and five have negative deltas in all four configurations, while the correlation of skill-level deltas between any two configurations ranges from -0.11 to 0.21. Even when the model is held fixed, the harness matters: with Kimi-K2.6, the observed deltas of 40 of the 98 skills change sign between Codex and Claude Code, as shown in Figure 7.

The `vitest-unit-testing` case in Figure 6 illustrates why the same skill can have different effects across configurations. The requirements are verified using the repository's native `vitest` framework, but the project dependencies are not installed in the initial checkout. The agent must therefore install the workspace before it can run the tests. Without the skill, Codex with GPT-5.4-mini installs the required toolchain in only four of the ten requirements and otherwise submits code without running the native test suite. With the skill, it installs the workspace and runs the relevant tests successfully in all ten requirements. The skill consequently receives a Golden classification under this configuration and produces the largest configuration-specific improvement reported in Table 3.

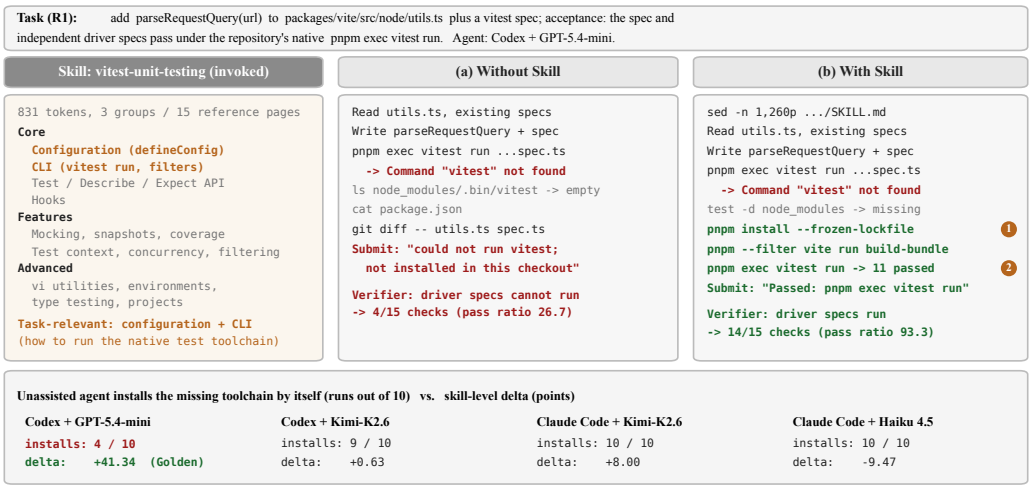

Figure 6: Case study of `vitest-unit-testing` (Codex with GPT-5.4-mini, R1, 26.7 $\rightarrow$ 93.3). **(a)** Without the skill, the agent hits `Command "vitest" not found` and submits untested code. **(b)** With the skill, it ① installs the dependencies and ② runs the tests to green. However, other configurations already install the toolchain on their own, so the skill adds little.

The other three configurations already install the missing toolchain in nine or ten of the unassisted runs. The skill therefore provides no consistent additional benefit and remains Ineffective under these configurations. In this case, the benefit comes primarily from changing the agent's setup and verification workflow rather than supplying task-specific testing knowledge. A skill's measured utility is thus the marginal difference between the behavior it induces and the behavior the configuration already exhibits, rather than a fixed property of the skill itself.

5 Discussion & Future Directions

Our results imply that skill utility is not an intrinsic property of a skill but a marginal effect relative to the default behaviour of an agent–model configuration. A skill helps when it supplies a concrete procedure the unassisted agent would otherwise omit, as with `viper-configuration` and `vitest-unit-testing`. It hurts when its concrete templates override the task's own instructions, as with `github-actions-templates`, or when broad, poorly matched guidance degrades performance across configurations, as with `asyncio-patterns`. Practically, skill authors should favor compact, operational guidance over hard-coded templates, and users should evaluate a skill with the model, agent, and repositories in which it will be deployed, since performance under one configuration does not prove universal utility.

Some limitations remain. The requirements are synthesized from each skill, so they test its claimed capabilities rather than the natural distribution of engineering requests. Our requirement-driven verification avoids LLM judging but may miss intended behaviors or favor particular implementations. Future work will audit verifiers through human review, repeat each evaluation, increase the number of requirements per skill to raise statistical power, and extend the benchmark to additional models, harnesses, and multi-skill settings.

6 Conclusion

We introduced SWE-Skills-Bench, a skill-centric framework for evaluating the marginal utility of agent skills through repository-grounded requirements, deterministic verifiers, and paired executions. Our findings show that providing a skill does not necessarily improve an agent's performance. Its value depends on whether it supplies procedures missing from the agent's default behaviour, while templates that override the task's own instructions may instead interfere with task execution. Skill utility should therefore be treated as configuration-specific rather than as an intrinsic property of the skill. SWE-Skills-Bench provides a practical foundation for evaluating skills with the models, agent harnesses, and task settings in which they will be deployed.

STATEMENTS

REPRODUCIBILITY STATEMENT

We support reproducibility through a fully specified, requirement-driven evaluation protocol. The benchmark contains 98 public software-engineering skills, each paired with a real GitHub repository pinned to a fixed commit. For every skill, we construct ten repository-specific requirements and translate their acceptance criteria into deterministic, execution-based verifiers. Both the skill-free and skill-assisted conditions use the same repository state, requirement, verifier, container environment, resource limits, command timeouts, and agent–model configuration; the only intended difference is access to the skill under evaluation. The paper reports the complete experimental protocol, statistical tests, timeout treatment, sensitivity analyses, construction prompts, and all skill-level results. The accompanying artifact is available at https://anonymous.4open.science/r/SWE-Skills-Bench. Each experimental cell currently contains a single paired execution, and this limitation is explicitly discussed in the paper.

AI USE STATEMENT

Generative AI systems were used both as objects of evaluation and as tools in the benchmark-construction pipeline. The evaluated software-engineering agents use language models to complete repository-grounded requirements with and without access to the target skill. Fixed prompts were also used to generate skill-specific requirements and to translate their acceptance criteria into executable verifiers. These generated artifacts were evaluated through deterministic, execution-based tests rather than an LLM judge. Generative AI tools additionally assisted with grammar, phrasing, and readability during manuscript preparation. The authors designed the study, selected the skills and repositories, defined the experimental and statistical protocols, analyzed the results, and reviewed the manuscript. The authors take full responsibility for the reported results, scientific claims, and final content.

ETHICS STATEMENT

This study evaluates software-engineering agents on public skills and public GitHub repositories and does not involve human participants, user studies, or deployment to real users. The research does not attempt to evaluate or profile repository contributors; repositories serve only as software environments for constructing and executing requirement-based tasks. All agent-generated modifications are evaluated offline in isolated, reproducible containers using deterministic checks. Because generated code and procedural guidance may be incomplete, insecure, or incompatible with a target project, benchmark outcomes should not be interpreted as certifying the correctness, security, or general suitability of any skill. Users of the released artifacts remain responsible for inspecting generated code and complying with the licenses and usage conditions of the underlying skills, repositories, models, and agent systems.

REFERENCES

Anthropic. Claude code: An agentic coding tool. GitHub, 2025a. 2025b. 5

Anthropic. Equipping agents for the real world with agent skills. Anthropic Engineering Blog, 2025b. 2025a. 1

Barry W. Boehm. *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, NJ, 1981. 2

Jun Shern Chan et al. MLE-bench: Evaluating machine learning agents on machine learning engineering. In *International Conference on Learning Representations (ICLR)*, 2025. 2

Mark Chen et al. Evaluating large language models trained on code. In *arXiv preprint arXiv:2107.03374*, 2021. 2, 3

Runnan Fang, Yuan Liang, Xiaobin Wang, Jialong Wu, Shuofei Qiao, Pengjun Xie, Fei Huang, Huajun Chen, and Ningyu Zhang. Mem^p: Exploring agent procedural memory. *arXiv preprint arXiv:2508.06433*, 2025. 1

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. In *International Conference on Learning Representations (ICLR)*, 2025. 2

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 (ICLR)*, 2024. 1, 3

Yuhang Lai et al. DS-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning (ICML)*, 2022. 2

Xiangyi Li, Yimin Liu, Wenbo Chen, et al. SkillsBench: Benchmarking how well agent skills work across diverse tasks. *arXiv preprint arXiv:2602.12670*, 2026. 2, 3

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. AgentBench: Evaluating LLMs as agents. In *International Conference on Learning Representations (ICLR)*, 2024. 2

Mike A. Merrill et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026. 2, 3

Bashar Nuseibeh and Steve Easterbrook. Requirements engineering: A roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pp. 35–46. ACM, 2000. 2

Klaus Pohl. *Requirements Engineering: Fundamentals, Principles, and Techniques*. Springer, Heidelberg, 2010. 2

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. 1

Ian Sommerville. *Software Engineering*. Pearson Education, 10th edition, 2015. 2

Wei Song, Haonan Zhong, Ziqi Ding, Jingling Xue, and Yuekang Li. Help or hurdle? rethinking model context protocol-augmented large language models. *arXiv preprint arXiv:2508.12566*, 2025. 1

Harsh Trivedi et al. AppWorld: A controllable world of apps and people for benchmarking interactive coding agents. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2024. 2

Vercel. skills.sh: The agent skills directory. https://skills.sh/, 2026. Accessed: 2026-09-25. 1

Virtual Uncle. Agent skills marketplace: Skills.sh guide (2026). https://virtualuncle.com/agent-skills-marketplace-skills-sh-2026/, 2026. Accessed: 2026-09-25. 1

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research*, 2024a. 1

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Ji-Rong Wen. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345, 2024b. 1

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hou, Yu Sun, Zhichao Yin, Ruilin Li, Heng Ji, and Graham Neubig. OpenHands: An open platform for AI software developers as generalist agents. In *International Conference on Learning Representations (ICLR)*, 2025. 1

Yaxiong Wu and Yongyue Zhang. Agent skills from the perspective of procedural memory: A survey. *TechRxiv preprint*, 2026. 1

Renjun Xu and Yang Yan. Agent skills for large language models: Architecture, acquisition, security, and the path forward. *arXiv preprint arXiv:2602.12430*, 2026. 1

John Yang, Akshara Prabhakar, et al. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. 1

John Yang, Kilian Lieret, et al. SWE-smith: Scaling data for software engineering agents. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025. 1

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023. 1

Shunyu Yao et al. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. In *International Conference on Learning Representations (ICLR)*, 2025. 2

Pamela Zave. Classification of research efforts in requirements engineering. *ACM Computing Surveys*, 29(4):315–321, 1997. 2

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. AutoCodeRover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2024. 1

Terry Yue Zhuo et al. BigCodeBench: Benchmarking code generation with diverse function calls and complex instructions. In *International Conference on Learning Representations (ICLR)*, 2025. 2

Table 3: Utility profiles of the nine skills that receive a Golden or Harmful label in at least one overall or configuration-specific assessment. Entries are skill-level deltas in points over the same ten requirements; * marks $p < 0.05$. Overall deltas average each requirement across the four configurations. Green and red mark observed direction only.

	Codex		Claude Code		
Skill	**GPT-5.4-mini**	**Kimi-K2.6**	**Kimi-K2.6**	**Haiku 4.5**	**Overall**
`viper-configuration`	+10.30	+17.33	+21.09*	+22.04*	+17.69*
`go-testify-patterns`	0.00	+2.16	+9.29	+8.05	+4.88*
`junit5-testing`	+2.58	+0.79	+1.68	+10.53*	+3.90*
`fastapi-patterns`	+10.33	−0.71	+3.85	−0.57	+3.23*
`vitest-unit-testing`	+41.34*	+0.63	+8.00	−9.47	+10.12
`drizzle-orm-patterns`	+9.29	+6.07	+0.72	+10.17*	+6.56
`asyncio-patterns`	−16.32	−18.61	−16.12	−16.79	−16.96*
`grafana-dashboards`	+0.36	−8.63	−2.08	−1.49	−2.96*
`statsmodels-regression`	+8.72	−18.60	+26.46	−14.65*	+0.48

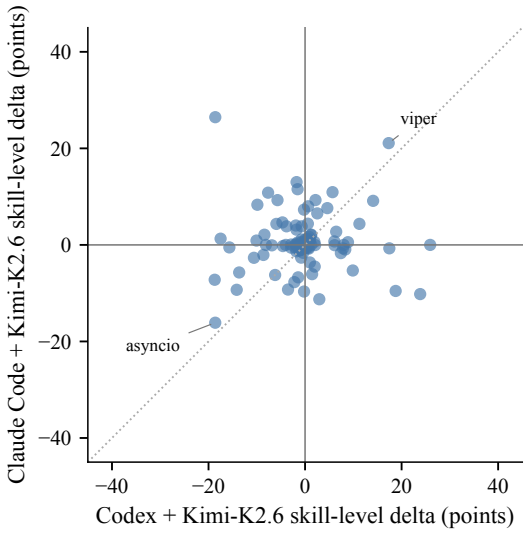

Figure 7: Skill-level deltas under the shared Kimi-K2.6 model, with each skill averaged over the same ten requirements in both harnesses. Axes share a scale; the diagonal marks equal effects. The plot describes observed effects, not significant interactions.

A Complete Results and Sensitivity Analyses

Table 3 lists every skill receiving a Golden or Harmful label in any overall or configuration-specific assessment; Figure 7 compares all skills across the two Kimi-K2.6 harnesses, and Figure 8 shows all ten paired requirement pass ratios for two high-magnitude cases (`vitest-unit-testing` and `asyncio-patterns`). The full 98-skill profiles appear in Figure 9 and Figure 10. All views use the same underlying experimental runs; the additional plots and subset analyses do not constitute independent replications.

A.1 Statistical and Timeout Sensitivity

Table 4 compares the primary exact sign-flip test with the two sensitivity tests specified in Section 3.4. All three tests yield the same four Golden and two Harmful skills. Table 5 examines the protocol decision to score timeout-terminated runs as zero. Removing the nine pairs that contain such a run changes the aggregate delta by only 0.02 points and leaves the overall classification unchanged.

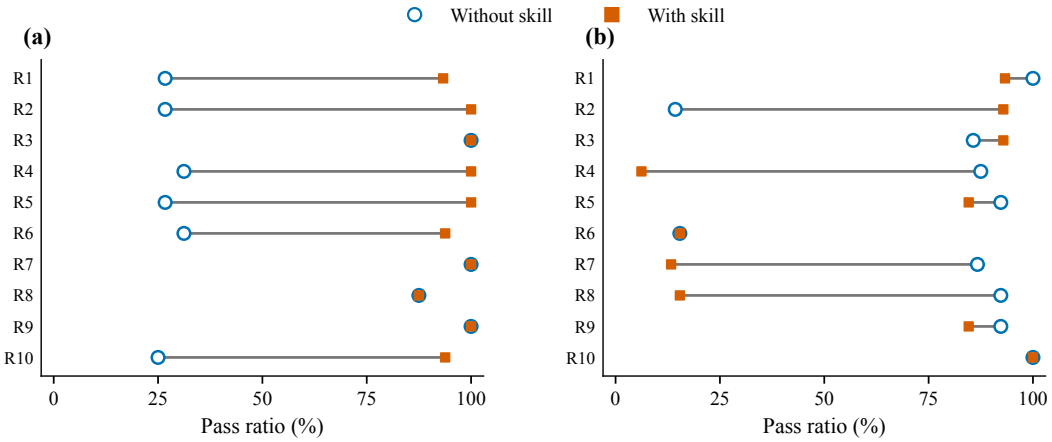

Figure 8: All ten paired requirement pass ratios for two selected high-magnitude diagnostic cases: (a) `vitest-unit-testing` with Codex and GPT-5.4-mini; (b) `asyncio-patterns` with Claude Code and Haiku 4.5. Lines connect pass ratios for the same requirement, with circles denoting no skill and squares denoting skill access. These candidates illustrate improvements and regressions, not representative population effects or established causal mechanisms.

Table 4: Sensitivity of overall classifications to the paired test. All 98 skills have complete ten-requirement panels in all four configurations. G/H/I denote Golden, Harmful, and Ineffective, respectively.

Paired test	**G**	**H**	**I**
Exact sign-flip (primary)	4	2	92
Paired t (sensitivity)	4	2	92
Exact Wilcoxon (sensitivity)	4	2	92

Table 5: Sensitivity to timeout-terminated runs. Timeouts are scored as zero under the benchmark protocol; the second row instead removes every pair in which either condition was terminated by the time limit and repeats the overall classification. G/H denote Golden and Harmful counts under the primary test.

Subset	**Pairs**	**Without**	**With**	**Δ**	**G**	**H**
All pairs	3,920	76.59	76.93	+0.34	4	2
Excluding pairs with a timeout	3,911	76.71	77.06	+0.36	4	2

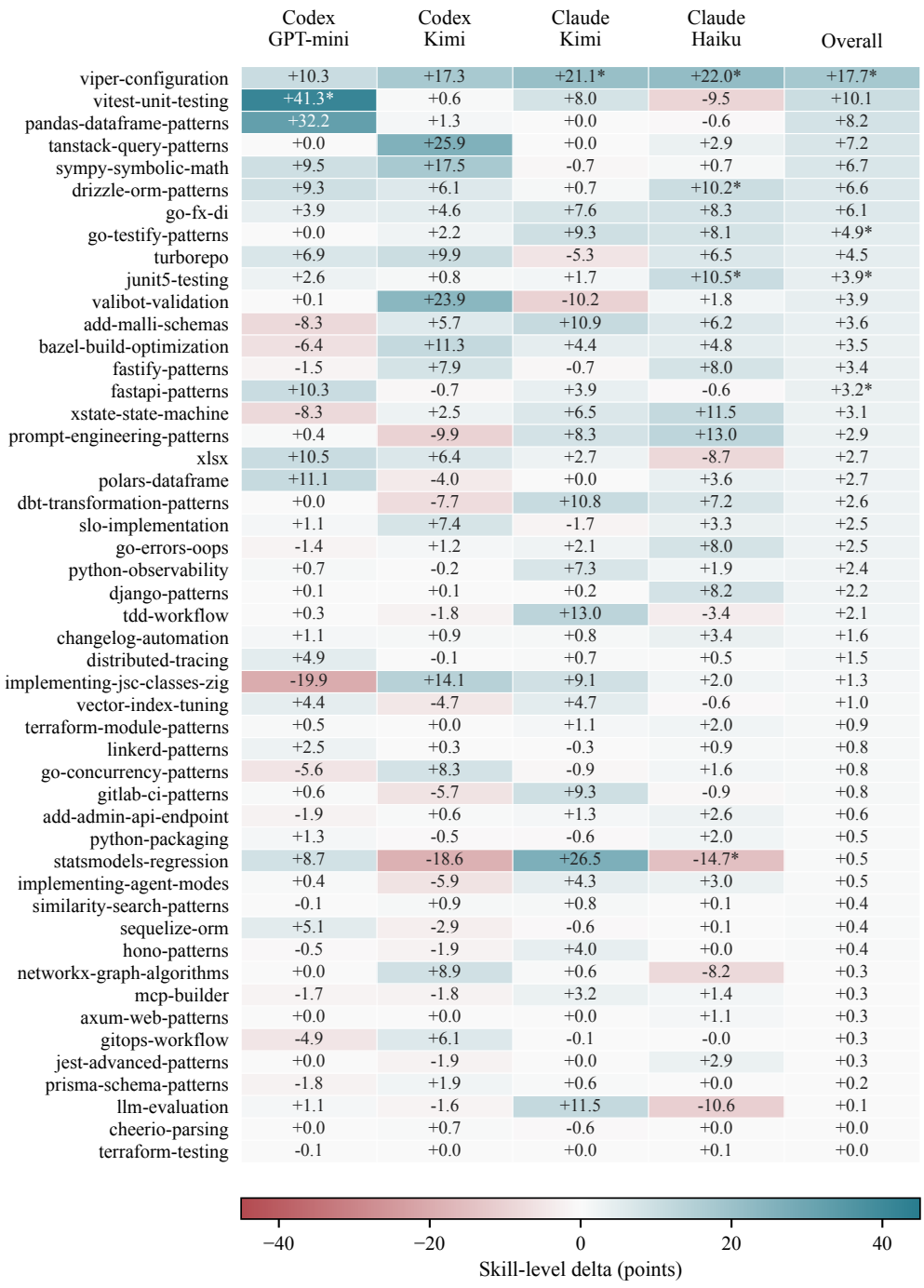

Figure 9: Full skill-level deltas, first 49 of 98 skills sorted by descending overall observed delta. Columns follow the four configurations in Table 2; GPT-mini denotes GPT-5.4-mini. Every entry averages the same ten requirements. Values are rounded to one decimal place. Stars mark $p < 0.05$. The same zero-centered color scale is used on both pages.

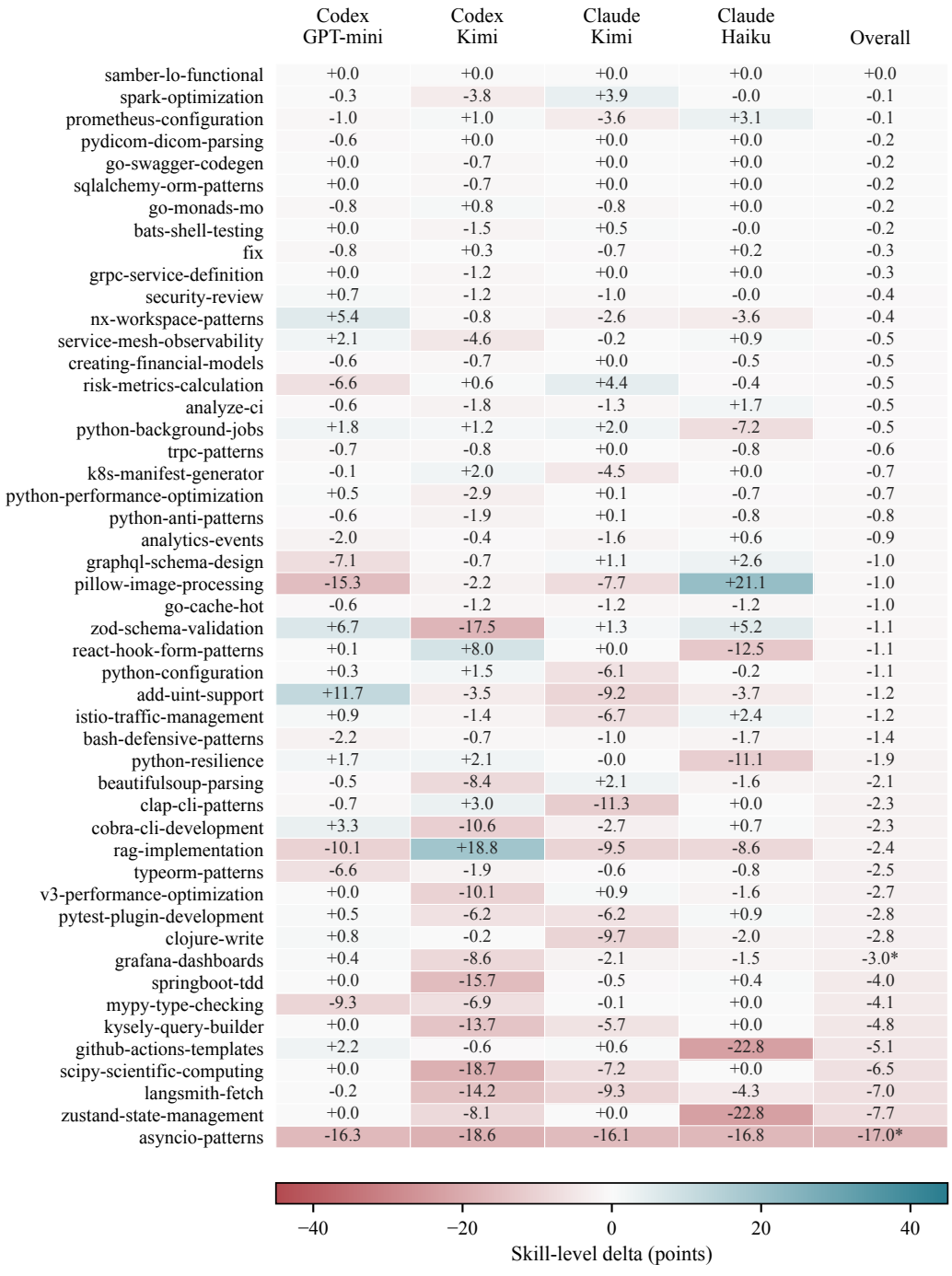

Figure 10: Full skill-level deltas, remaining 49 skills, continuing Figure 9 with the same ordering and color scale. Displayed near-zero deltas must not be interpreted as established equivalence. Stars denote $p < 0.05$ in that assessment.

System prompt

You are a professional test engineer. Your task is to generate automated test suites that evaluate whether an AI Agent has correctly completed a given programming task.

Context
An AI Agent is given a programming task to complete within a real software environment. After the Agent finishes, the test suite you generate is executed to automatically determine whether the task was completed correctly.
Core Principles
1. **Discriminative Power**: Tests must reliably distinguish between "genuinely completed" and "superficially plausible" outputs.
2. **Behavioral Verification**: Execute the code and verify its actual output — do not rely solely on static checks such as keyword matching or file existence.
3. **Completeness**: Cover all acceptance criteria specified in the task, including boundary conditions and error handling.
4. **Non-Trivial Assertions**: Verify the correctness of values, data structures, and logic — not merely their presence.

Test Design Guidelines
Required Strategies
- **Run and verify**: Execute the Agent's code and check return values, output content, and side effects.
- **Structural validation**: For generated configs, schemas, or data files, parse and validate the structure and semantics — not just syntax.
- **Edge-case testing**: Include tests for boundary inputs, missing fields, and expected error behaviors.
Prohibited Patterns
- Keyword-only assertions (e.g., `assert "keyword" in source_code`)
- Overly permissive checks (e.g., `assert len(results) >= 1`)
- File-existence-only checks without content verification
- Any assertion that a trivially incorrect output could pass
Output Requirements
- Generate ≥ 10 test cases per task, spanning multiple difficulty levels.
- Each test must include a docstring explaining what it verifies.
- Output only executable test code — no explanatory prose.

User Prompt

Please generate a test suite for the following AI Agent task.

Task ID
{task_id}
Environment
- Project root: {project_root}
- Language / Framework: {language}
- Available toolchain: {toolchain}
Task Description
{task_description}
Acceptance Criteria
{acceptance_criteria}
Constraints
- Per-test timeout: {timeout} seconds
- Available tools in the execution environment: {available_tools}

Figure 11: The prompt used for requirement-driven verification generation.

B Construction Prompts and Examples

 # Task Requirement Document Generator — Prompt Template
You are a ****task requirement document generator****. Generate a ****Task Requirement Document**** (Markdown) based on the provided {{information_sources}} and {{skill_reference}}. The task requirement should represent a realistic {{task_scope}} within the {{target_context}}.
The generated document must be ****self-sufficient****: a {{task_executor}} reading only this document must understand what to build, where to make changes, and what counts as done.
Core Principles
1. Match the {{target_context}}'s real ****{{tech_constraints}}****.
2. Stay within the ****problem scope**** defined by {{skill_reference}}.
3. Focus on concrete ****objectives, constraints, {{artifact_location}}s, and verifiable outcomes****.
4. ****Do not leak**** {{skill_reference}} methods, best practices, or implementation patterns.
Information Sources
- ****Configuration****: id, name, description, type, evaluation config from {{config_source}}
- ****Skill Reference****: {{{skill_reference}} full content}
- ****Task Template****: {{{task_template}} full content}
Required Sections
Section	Purpose
Background	Context a {{task_executor}} needs
{{artifact_section_name}}	Concrete {{artifact_type}} paths to create/modify
Requirements	Behavior, constraints, edge cases — ****not**** solutions
Acceptance Criteria	Observable, verifiable "done" conditions
Specificity Rules
- Name exact {{domain_specifics}} whenever part of the goal.
- Name exact {{artifact_type}} paths — avoid vague entries like "{{vague_example}}".
- Name important edge cases, validation rules, and failure modes.
Anti-Patterns
- ✗ Rewriting {{skill_reference}} steps/patterns into the task requirement.
- ✗ Guiding language ("recommended to use", "suggest adopting").
- ✗ Vague {{artifact_type}} locations or validation criteria.
- ✗ Technology stack mismatching the {{target_context}}.
Self-Check
- [ ] {{task_executor}} can complete without hidden knowledge.
- [ ] Every {{artifact_type}} entry is concrete and {{path_format}}.
- [ ] Acceptance Criteria are outcome-based.
- [ ] No {{skill_reference}} methodology leaked.

Template Variables
Variable	Description	Example
`{{task_executor}}`	Who executes the task	agent, developer
`{{target_context}}`	Project environment	repository, codebase
`{{skill_reference}}`	Knowledge doc (not to leak)	SKILL.md
`{{tech_constraints}}`	Technical boundaries	tech stack, testing style
`{{artifact_type}}`	Deliverable unit	file, component
`{{artifact_section_name}}`	Section header for deliverables	Files to Create/Modify
`{{path_format}}`	How paths are expressed	repository-relative
`{{domain_specifics}}`	Domain nouns to be precise about	APIs, schemas, events
`{{vague_example}}`	Anti-pattern example	"files under src/"
`{{config_source}}`	Configuration origin	benchmark_config.yaml
`{{task_scope}}`	What the task represents	development task
`{{information_sources}}`	Generator inputs	repo info, eval config

Figure 12: The prompt used for task instance requirement generation.

 # Task: Extend Unsigned Integer Type Coverage in PyTorch Operators
Background
The PyTorch codebase (https://github.com/pytorch/pytorch) has partial support for unsigned integer types across its operator library. Several arithmetic and mathematical operators currently lack support for `uint16`, `uint32`, and `uint64` tensor types. When users pass tensors of these dtypes to unsupported operators, the runtime raises a dispatch error. The type dispatch infrastructure needs to be extended so that additional operators can accept unsigned integer inputs.
Files to Modify
- `aten/src/ATen/native/BinaryOps.cpp` — Dispatch registration for arithmetic ops (add, sub, mul, floor_divide, remainder) and bitwise ops (and, or, xor, lshift, rshift)
- `aten/src/ATen/native/cpu/BinaryOpsKernel.cpp` — CPU kernel implementations for binary operators
- `aten/src/ATen/native/TensorCompare.cpp` — Dispatch registration for comparison ops (eq, ne, lt, le, gt, ge)
- `aten/src/ATen/native/GcdLcm.cpp` — GCD operator dispatch and implementation
Requirements
1. Operators to support
- The implementation MUST explicitly add `uint16`, `uint32`, and `uint64` support for the following operators (operator names correspond to native ATen symbols and Python API where applicable):
- Arithmetic: `add` (aten::add), `sub` (aten::sub), `mul` (aten::mul)
- Integer division / remainder: `floor_divide` (aten::floor_divide), `remainder` (aten::remainder)
- Bitwise and shifts: `bitwise_and` (aten::bitwise_and), `bitwise_or` (aten::bitwise_or), `bitwise_xor` (aten::bitwise_xor), `lshift`/`left_shift` (aten::lshift), `rshift`/`right_shift` (aten::rshift)
- GCD: `gcd` (aten::gcd) when present in the codebase
- Comparisons: `eq`, `ne`, `lt`, `le`, `gt`, `ge` (aten::* comparison ops)
2. Scope of changes
- For each operator above, update both the operator registration/type-dispatch tables and the CPU kernel implementations under `aten/src/ATen/native/` so that the operator accepts unsigned integer tensors without raising dispatch errors.
- If a kernel implementation is missing for an unsigned dtype, add an explicit kernel path (reuse signed-integer implementation where semantics match, or add a thin adapter that performs identical, dtype-preserving logic).
- Do NOT change operators that are inherently floating-point-only (e.g., `sqrt`, `sin`, `exp`).
3. Semantics and dtype rules
- When all inputs are the same unsigned integer dtype, the operator should preserve that dtype for outputs where that is semantically correct (e.g., `add(uint32, uint32) -> uint32`).
- For mixed-type inputs, follow existing PyTorch promotion rules; do not introduce new promotion behavior beyond existing signed-integer promotion rules.
- Integer division behavior: implement `floor_divide` and `remainder` semantics consistent with current PyTorch integer ops (no conversion to floating point), and ensure results for unsigned inputs match the mathematical remainder/quotient for non-negative integers.
4. Compatibility and robustness
- Ensure changes compile and pass existing unit tests unrelated to unsigned support.
- Provide fallbacks or clear error messages for operator combinations that remain unsupported (e.g., mixing unsigned with types that cannot be sensibly combined).
5. Implementation notes for contributors
- Prefer adding dtype coverage via dispatch table entries and small adapters rather than rewriting algorithmic kernels.
- Include small unit tests for each operator (see Acceptance Criteria) to validate behavior on sample inputs.
Expected Functionality
- Operators that previously raised dispatch errors for `uint32` or `uint64` tensors now execute successfully and return correct results
- The output tensor dtype matches the input tensor dtype when all inputs share the same unsigned type
- Operators that only make sense for floating-point types remain unchanged
Additionally, the repository should include minimal unit tests that demonstrate correct behavior for each operator listed in the Requirements (examples in Acceptance Criteria). These tests should validate dtype preservation, numerical correctness for representative values, and reasonable behavior for edge cases (e.g., zero, max-value, boundary shifts).
Acceptance Criteria
- The listed operators accept `uint16`, `uint32`, and `uint64` inputs without raising dispatch errors.
- Arithmetic, division, remainder, bitwise, GCD, and comparison results are numerically correct for representative unsigned inputs.
- When all inputs share the same unsigned dtype, the result keeps that dtype wherever PyTorch's existing promotion rules do not require otherwise.
- Edge cases including zero values, maximum representable values, and boundary shift counts behave consistently and do not crash.
- Floating-point-only operators remain unchanged and unsigned support is limited to operators with well-defined integer semantics.

Figure 13: An example of the generated requirement in SWE-Skills-Bench.