SKILLGENBENCH: BENCHMARKING SKILL GENERATION PIPELINES FOR LLM AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

As LLM agents are increasingly built around reusable *skills*, a central challenge is no longer only whether agents can *use* provided skills, but whether they can *generate* correct, reusable, and executable skills from repositories and documents. Existing benchmarks primarily evaluate the efficacy of given skills or the ability of agents to solve downstream tasks from raw context, but they do not isolate *skill generation* itself as the object of study. We introduce **SkillGenBench**, a 211-task benchmark for evaluating skill generation pipelines under a unified and controlled protocol. In SkillGenBench, a generator receives raw corpora and produces standardized skill artifacts, which are then executed under fixed harnesses and assessed with unified evaluation procedures. The benchmark covers two generation regimes: *task-conditioned generation*, where a task-specific skill is synthesized after the task is revealed, and *task-agnostic generation*, where a reusable skill library must be distilled before downstream tasks are known. It also spans two complementary procedural sources: *repository-grounded* instances, where procedures are distributed across code, configuration, and scripts, and *document-grounded* instances, where procedures and constraints must be distilled from long-form text. We provide standardized task specifications, pinned environments, and evaluation protocols centered on deterministic execution-based checks, supplemented by auxiliary signals for diagnosis. Experiments across a range of skill-generation methods and backbones show substantial performance variation, highlight the difficulty of reusable skill distillation, and reveal distinct failure modes in skill generation from software repositories versus long-form documents. SkillGenBench establishes a reproducible testbed for studying skill generation as an independent research problem in agent systems.

1 INTRODUCTION

As LLM agents are deployed in increasingly complex environments, a growing design trend is to move beyond monolithic prompting and toward modular, persistent capability abstractions. Recent agent systems increasingly rely not only on runtime augmentation through tools and external context, but also on reusable *skills*: packaged procedural artifacts that encode how to accomplish classes of tasks in a form that can be stored, versioned, and reused. In emerging Agent Skills interfaces Anthropic (2025), a skill is typically organized around a `SKILL.md` file together with optional scripts, references, and auxiliary resources. Skills can be audited, cached, shared, and updated independently of the base model, making procedural knowledge reusable across executions.

Early empirical evidence highlights both the promise and the fragility of skill-based agent design. *CL-Bench* Dou et al. (2026) shows that even when relevant evidence is explicitly present in complex context, models frequently fail to extract and operationalize it into correct procedures. In parallel, *SkillsBench* Li et al. (2026b) shows that curated skills can substantially improve downstream task performance, while automatically generated skills—especially those produced on the fly—are often unstable and can even induce negative transfer. Taken together, these findings suggest a broader lesson: procedural knowledge is valuable when externalized into structured artifacts, but difficult for models to reliably distill from raw repositories, documents, and other unstructured corpora.

This tension becomes more important in realistic deployment settings, where procedural knowledge is not static. New repositories, APIs, technical documents, and papers continuously introduce

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.

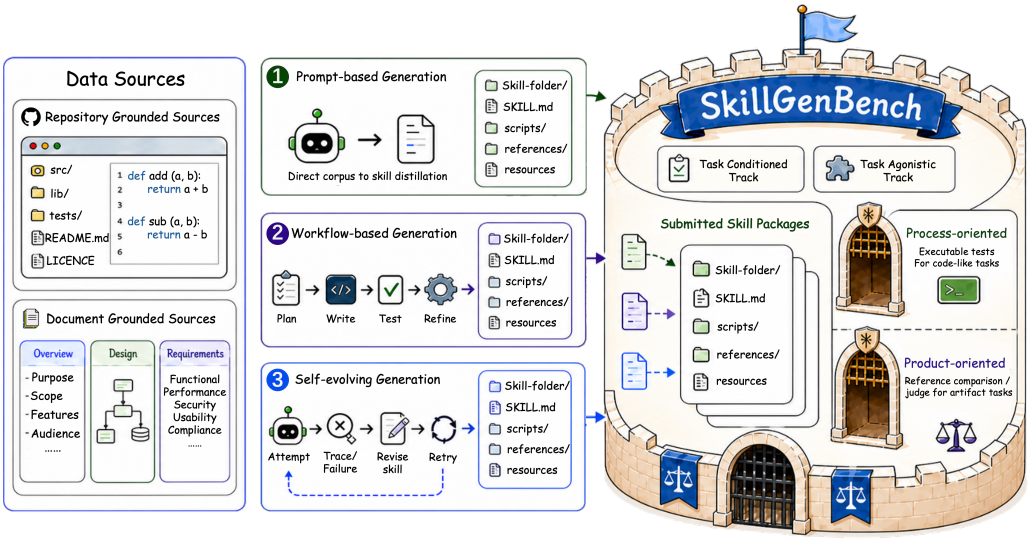


Figure 1: Overview of SkillGenBench. Skill-generation pipelines transform repository- and document-grounded sources into standardized skill packages, which are evaluated under task-conditioned and task-agnostic tracks with fixed execution checks and artifact-level diagnostics.

new constraints, workflows, and best practices that must be incorporated if agents are to remain current (Liang et al., 2026). In such settings, the central challenge is not only whether an agent can *use* a provided skill, but whether a pipeline can *generate* a correct, reusable, and executable skill from visible corpora. Yet existing benchmarks rarely isolate this generation step as the primary object of evaluation. Skill-centric benchmarks (Li et al., 2026b; Liu et al., 2026) typically measure whether a provided skill improves downstream execution; task-centric benchmarks (Liu et al., 2024; Zhou et al., 2024; Jimenez et al., 2023; Merrill et al., 2026; Dou et al., 2026) measure whether an agent can solve an end task from raw context. Neither offers a controlled protocol for comparing *skill generation pipelines* as modular, interchangeable components under fixed downstream execution conditions.

We address this gap with **SkillGenBench**, a 211-task benchmark for evaluating **skill generation pipelines** under a unified and controlled protocol. As shown in Figure 1, SkillGenBench treats the generator itself as the object of study: given raw corpora, a generator produces standardized skill artifacts, which are then executed under fixed harnesses and assessed with a unified evaluation procedure. We study two practically important generation regimes. In **task-conditioned generation**, the generator synthesizes a task-specific skill from a raw corpus together with the task specification. In **task-agnostic generation**, the generator must distill a reusable skill library from raw corpora *before* downstream tasks are revealed. The resulting library is generated once and reused across downstream tasks, with its utility measured through execution outcomes.

SkillGenBench spans two complementary procedural sources. **Repository-grounded** instances require generators to recover procedures distributed across repository structure, code, configuration, and scripts. **Document-grounded** instances require generators to distill procedures and constraints from long-form knowledge sources whose relevant evidence may be explicit but dispersed. To enable reproducible comparison, we provide standardized task specifications, pinned environments, fixed execution harnesses, and unified evaluation protocols centered on deterministic execution-based checks, supplemented by auxiliary similarity-based and judge-based signals where needed for diagnosis (Li et al., 2026b; Jimenez et al., 2023; Merrill et al., 2026). Our contributions are: (1) a benchmark that directly evaluates skill generation pipelines, rather than provided skills or unconstrained end-to-end agents; (2) a task-agnostic setting that measures one-shot reusable skill library distillation before hidden downstream tasks are revealed; (3) a unified benchmark spanning both repository-grounded and document-grounded procedural knowledge; and (4) a reproducible empirical study across representative generator families with systematic failure analysis.

Table 1: Comparison with representative skill-related and agent benchmarks.

Dimension	SkillsBench	SWE-Skills	WildClawBench	SkillGenBench
Primary target	Skill efficacy	SWE skills	Agent ability	Skill gen.
Skill acquisition	Curated/self-gen.	Public SWE skills	Runtime capability	Self-generated
Task-conditioned generation	✓	✗	✗	✓
Task-agnostic generation	✗	✗	✗	✓
Process-oriented evaluation	✓	✓	✓	✓
Product-oriented comparison	✗	✗	✗	✓

2 Related Work

2.1 Agent Skills and Runtime Augmentation

A substantial body of work extends agent capability through runtime augmentation, including reasoning-and-acting loops (Yao et al., 2023), tool use (Schick et al., 2023; Qin et al., 2024), retrieval augmentation (Lewis et al., 2020), and standardized interfaces such as MCP (Anthropic, 2024). While effective, these approaches primarily improve what an agent can accomplish within a single execution episode, leaving procedural knowledge implicitly embedded in prompts, traces, or retrieved context.

Recent work increasingly treats *skills* as reusable procedural artifacts that persist beyond individual executions (Jiang et al., 2026). Early systems acquire or consolidate skills from agent experience without standardized packaging (Wang et al., 2023; Shinn et al., 2023; Zhao et al., 2024; Huang et al., 2025), whereas recent frameworks adopt explicit skill abstractions with portable packaging interfaces (Anthropic, 2025). Building on this abstraction, subsequent work studies skill creation (Liang et al., 2026), orchestration and routing (Li et al., 2026a; Zheng et al., 2026), and reusable skill ecosystems for long-horizon agent workflows.

2.2 Skill Generation Pipelines

As skills are increasingly treated as first-class artifacts, *skill generation* has emerged as an important research direction that distills procedural knowledge from repositories, documentation, papers, and agent experience into reusable skill artifacts. Existing methods broadly follow three patterns: extracting and distilling procedures from corpora into structured skills Liang et al. (2026), experience-driven consolidation from successful interactions or trajectories (Wang et al., 2026; Yang et al., 2026; Ni et al., 2026), and iterative refinement through execution feedback or structural validation (Zheng et al., 2025; Alzubi et al., 2026; Xia et al., 2026; Ma et al., 2026; Lu et al., 2026; Zhou et al., 2026). However, these generators are typically evaluated together with bespoke executors, routing policies, retrieval configurations, and environment assumptions. This coupling makes it difficult to disentangle the quality of skill generation from downstream integration choices. As a result, the field still lacks a benchmark that compares skill generation pipelines themselves as modular and interchangeable components under a common downstream protocol.

2.3 Skill Benchmarks

Recent skill benchmarks (Li et al., 2026b; Liu et al., 2026; Ding et al., 2026) primarily evaluate *skill efficacy*, assessing whether a provided skill artifact improves downstream execution for a given task under controlled evaluation settings. SkillsBench Li et al. (2026b) compares the settings of no-skill, curated skill, and self-generated skills in domains under deterministic verifiers, and shows that curated skills can be beneficial while self-generated skills can be unstable. SWE-Skills-Bench Han et al. (2026) applies the same paired-evaluation logic to software engineering by pairing collected public skills with real-world repositories pinned at fixed commits with requirement-driven, execution-based verification. Although valuable, these benchmarks do not systematically evaluate *skill generation pipelines*. As summarized in Table 1, existing benchmarks differ in domain and execution environment, but they do not cover a task-agnostic library setting in which reusable skills must be distilled before hidden tasks are revealed. In SkillsBench, the self-generated condition is task-local: skills are generated only after the task is revealed and consumed immediately, rather than being compared under a common protocol across dedicated generators. SkillGenBench complements

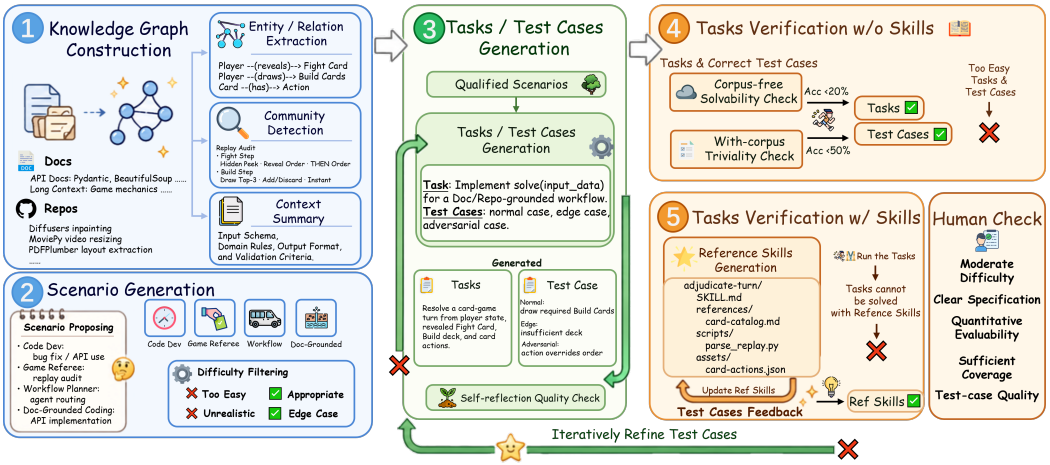


Figure 2: Automated task-construction pipeline in SkillGenBench. Repositories and long documents are first abstracted into a knowledge graph (Stage 1). Task scenarios are then proposed and filtered (Stage 2), and each scenario produces tasks and their test cases (Stage 3). Stage 4 filters out tasks solvable without procedural extraction or trivially solvable with the full corpus, and Stage 5 validates the remaining tasks with an iteratively refined reference skill. Tasks failing Stage 4 or Stage 5 are returned to Stage 3 for test-case refinement. Accepted tasks finally undergo human verification.

this line of work by treating skill generation pipelines as the primary object of evaluation, comparing pipelines that transform visible corpora into reusable skill artifacts and testing those artifacts under fixed harnesses and deterministic verification.

3 SkillGenBench

SkillGenBench evaluates how LLMs distill procedural knowledge from repositories and documents into reusable skills. A generator produces a skill package, and a separate executor uses it to complete downstream tasks. Holding the execution configuration fixed within comparisons lets us study how generation pipelines affect task performance.

At the instance level, each benchmark item is packaged as a containerized environment comprising five components: source materials, task specification, skill interface, executor, and evaluation protocol.

3.1 Sources of Procedural Knowledge

SkillGenBench instances are organized by the source of procedural knowledge: *repository-grounded* and *document-grounded*. The two differ in how procedures are presented in the source materials and, accordingly, in what the model must extract.

Repository-grounded instances. The source materials consist of a code repository snapshot, including directory structures, README files, configuration files, dependency scripts, and environment conventions. Procedural knowledge is rarely stated explicitly; instead, it is implicit in code organization, call relations, entry scripts, and runtime constraints. The model must recover these latent workflows from the repository and distill them into a reusable skill.

Document-grounded instances. The source materials consist of dense, long-form texts such as system manuals, API specifications, and technical reports. In contrast to repository-grounded instances, procedural knowledge is expressed explicitly but distributed across passages, taking forms such as conditional branches, parameter rules, prerequisites, and ordered steps. The model must integrate these scattered constraints into a single skill that can be invoked on downstream tasks.

For source-level analysis, we distinguish repository snapshots, software references, and domain documents. Software references cover API and tool documentation as well as selected implementation excerpts; domain documents emphasize rule application and exact output constraints. SkillGenBench contains 211 task instances, including structured domain reasoning, media processing, document workflows, and native visual interpretation. Figure 3 summarizes the sources and task families across the benchmark; Appendix A.1 specifies task coverage and execution configurations.

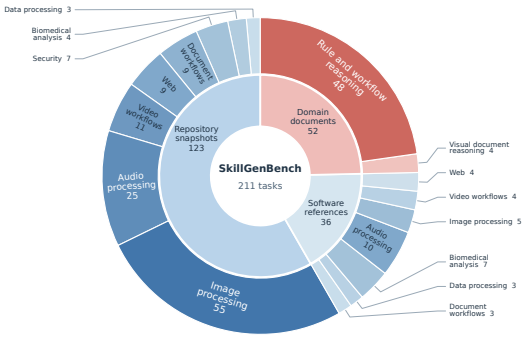


Figure 3: Source types (inner ring) and task families within each source (outer ring) across the 211-task benchmark. Numbers denote task counts.

3.2 Skill Generation Settings

SkillGenBench defines two task settings based on whether the downstream task is known at generation time. The *task-conditioned* setting reveals the task to the model; the *task-agnostic* setting does not.

Task-conditioned setting. The model receives the source materials together with a task specification, and must identify the procedures most relevant to the task and distill them into a focused skill. This setting evaluates targeted distillation: whether the model can filter out irrelevant information and recover the key procedure required by the task.

Task-agnostic setting. The model receives only the source materials, with no access to downstream tasks. It must build a reusable skill library within a fixed generation budget; this library is then used to support held-out tasks revealed at execution time. The challenge here is not task-specific synthesis but the identification of procedures with cross-task reuse value, and their organization into deployable skills without task hindsight.

3.3 Benchmark Construction Pipeline

Automated candidate construction follows the pipeline shown in Figure 2. We collect two classes of source materials: fixed-commit repository snapshots and long-form document bundles. For repository-grounded instances, we prioritize repositories in which key procedures are distributed across code, configurations, scripts, and environment conventions. For document-grounded instances, we prioritize long documents whose procedural constraints span multiple sections and cannot be recovered from a single passage.

Stage 1: Knowledge Graph Construction. We construct knowledge graphs to support the subsequent stages. Each graph abstracts the raw corpus into entity-relation triples, communities of related procedural evidence, and context summaries covering input schemas, domain rules, output formats, and validation criteria.

Stage 2: Scenario Generation. From the knowledge graph and context summaries, we derive candidate scenarios across several common task forms, such as code development, workflow execution, and rule-grounded reasoning. Each scenario identifies a target workflow and the relevant corpus evidence.

Stage 3: Tasks and Test Cases Generation. Each scenario is used to generate a task specification and a set of test cases covering normal, edge, and adversarial inputs. A self-reflection step then refines each candidate for clarity and consistency.

Stage 4: Task Verification without Skills. We discard tasks that are either solvable without procedural extraction or trivially solvable. Specifically, we run two checks with a strong base model (e.g., GPT-5): a *corpus-free check*, where the model attempts the task using only its parametric knowledge, and a *with-corpus check*, where the model is given the full source materials. Tasks with a

pass rate $\geq 20\%$ on the corpus-free check or $\geq 50\%$ on the with-corpus check are returned to Stage 3 for refinement.

Stage 5: Task Verification with Skills. For each remaining task, we generate a reference skill and refine it through iterative test-case feedback. We then run the task using this skill. If the task fails to pass even with the reference skill, it is judged unrealistic or overly hard, and is sent back to Stage 3.

This process repeats until the task falls within the target difficulty range or reaches the iteration limit. The benchmark contains 211 tasks reviewed for specification clarity, source alignment, and verifier consistency (Appendix F.1). Candidates that fail validation are refined or replaced. The retained tasks cover repository procedures, structured domain reasoning, media and document workflows, and visual interpretation, with instance-specific verifiers.

3.4 Evaluation Protocol

SkillGenBench evaluates a generated skill by its downstream behavior. An executor loads the skill and attempts the task. Instances use either *execution-based* or *artifact-based* verification.

During skill generation, the model has access to the source materials and, in the task-conditioned setting, the task specification. Test cases, verifier internals, reference outputs, and held-out tasks are never exposed to the model. During execution, generated skills are run in containerized environments under the executor assigned to the task configuration.

Execution-based evaluation. The submitted code is run against hidden test cases with deterministic expected outputs, analogous to program-judging benchmarks. This mode is used when the desired result is a callable procedure or reusable implementation.

Artifact-based evaluation. The submitted code is first executed to produce an artifact, which is then compared against a reference output. Comparison methods depend on the output modality, including exact matching, pixel-level similarity, semantic similarity, or an LLM judge when multiple valid outputs cannot be captured by a single deterministic metric. A heuristic pre-check (for example, resolution, duration, schema, or file format) filters out invalid outputs before comparison. This mode does not assume a unique ground-truth implementation, since many tasks admit multiple valid programs producing equivalent outputs.

4 Experiments

4.1 Experimental Setup

We evaluate five skill-generation method families across six generator backbones: Claude Sonnet 4.5 Anthropic (2025b), GPT-5 Singh et al. (2025), Kimi K2.5 Team et al. (2026), GLM-5 Zeng et al. (2026), MiniMax-M2.7 MiniMax (2026b), and Qwen3.6-Plus Qwen Team (2026). All agentic interactions use the Claude Code runtime Anthropic (2025a); task success is determined by the instance-specific verifier. Baseline implementations are described in Appendix C.

For Table 2, the executor is fixed within each task configuration: MiniMax-2.5 MiniMax (2026a) uses a 1800-second per-instance limit, and GPT-5.5 uses 600 seconds and 40 turns per trial. Method adaptations, including source-only two-stage EvoSkill and SkillCreator v2 in the GPT-5.5 configuration, are specified in Appendices B and C.

We report **pass@3**: a task is solved if any of up to three trials passes, with early stopping after success. Table 2 aggregates successes over evaluable tasks; missing packages and unavailable model calls are reported separately. Task coverage and execution settings vary across cells, so method comparisons use matched task sets (Appendix A.1). We also compare GPT-5.5 and GLM-5.3 with generated skills held fixed.

Table 2: Pass@3 (%) by method and generation backbone over evaluable tasks. Coverage and execution configurations vary across cells (Appendix A.1).

Method	Sonnet 4.5	GPT-5	Kimi K2.5	GLM-5	MiniMax M2.7	Qwen3.6 Plus
NAIVE PROMPT	19.0	21.8	12.3	24.9	21.8	24.2
EVOSKILL	20.7	25.2	10.2	14.3	14.9	22.1
SKILLNET	17.5	25.0	12.8	20.6	14.6	25.6
SKILLCREATOR	17.0	23.2	12.8	27.0	18.5	24.6
SKILLSEEKERS	24.6	28.9	14.4	26.1	24.2	26.1

4.2 DYNAMIC EXECUTION RESULTS

Table 2 summarizes the available task outcomes for all five method families and six generation backbones. The shared no-skill reference achieves 20.9% over the full task inventory; Appendix A.1 also gives controls restricted to each cell's evaluated tasks.

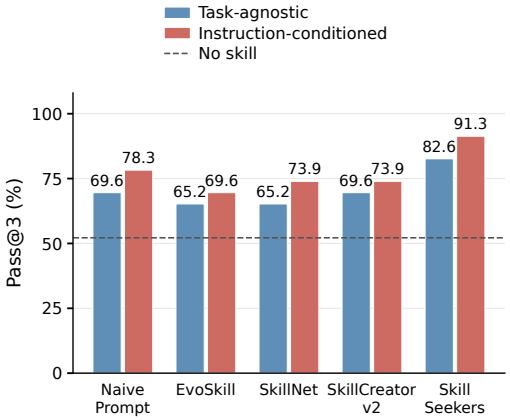


Figure 4: Task-agnostic versus public-instruction-conditioned pass@3 across five methods, using GPT-5.5 for generation and execution. The dashed line marks no skill. EvoSkill uses the spec-only adaptation.

Method comparison. For the GPT-5 generator under GPT-5.5 execution, the five methods share 20 evaluable tasks. Naive Prompt, EvoSkill, SkillNet, SkillCreator v2, and SkillSeekers solve 16, 15, 16, 16, and 19 tasks, respectively; no skill solves 12 and full source solves 19. Thus, SkillSeekers matches direct source access on this subset, while all five methods improve on its no-skill control.

Source difficulty remains visible in the configuration-specific breakdown (Appendix Table 6). Repository-task averages range from 10.8% to 14.4%, compared with 21.4% to 25.0% for documentation tasks. Repository-grounded generation requires recovering environment setup, command conventions, and data flow from distributed code artifacts. A skill can capture the source vocabulary while missing the executable procedure needed by the verifier.

Generation regime. Figure 4 compares source-only and public-instruction-conditioned generation using GPT-5.5 for both generation and execution. The conditioned generators see the sources and public instructions, without task inputs, answers, or evaluation feedback. Of 24 tasks, 23 have valid packages under both regimes for all five methods. Conditioning adds one or two successful tasks per method (+4.35 to +8.70 percentage points). The paired comparison reuses nonconcurrent runs, and method-matched source-cluster intervals include zero; Appendix B gives the matching protocol and generation coverage.

Cross-executor evaluation. We compare GPT-5.5 and GLM-5.3 on 20 shared benchmark tasks, using the same GPT-5.5-generated, source-only skills, task inputs, and trial budgets. No Skill, Full Source, and Naive Prompt attain 55%, 85%, and 75% pass@3 with either executor; EvoSkill attains 65% and 60%, respectively. Both skill methods exceed the no-skill aggregate while falling below direct source access on this subset. Appendix A.4 reports the task coverage and paired outcomes.

4.3 STATIC ANALYSIS

Static diagnostics characterize the artifacts produced by each pipeline, complementing execution-based pass@3.

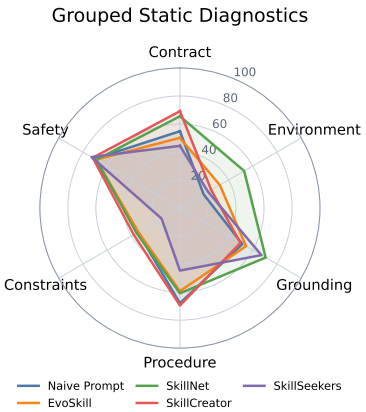


Figure 5: Grouped static diagnostics over generated skill files. Axes aggregate automatic rule-based checks; higher is better.

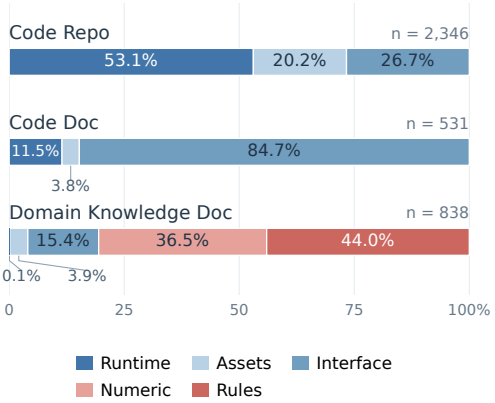


Figure 6: Completed verifier-failure composition by source. Segments show within-source percentages; n denotes classified failures.

Table 3: Grouped static skill scores by method. Scores average generated `SKILL.md` file records in the fixed six-backbone inventory on a 0–100 scale. Overall averages the six grouped diagnostics.

Method	Overall	Contract	Env.	Ground	Proc.	Const.	Safety
NAIVE PROMPT	49.8	54.8	19.5	51.5	67.9	34.6	70.3
EVOSKILL	50.0	50.1	32.9	54.5	59.3	34.2	69.0
SKILLNET	**59.1**	65.5	**52.9**	**70.7**	60.9	35.8	68.9
SKILLCREATOR	54.0	**69.1**	25.8	49.8	**69.5**	**38.1**	71.9
SKILLSEEKERS	44.6	44.4	23.2	67.2	44.6	15.1	**72.8**

We apply eight rule-based diagnostics to the fixed six-backbone inventory of generated `SKILL.md` files and group them into six axes. *Contract* averages interface-contract and verification cues. *Environment* measures setup and dependency readiness. *Grounding* measures explicit ties to source artifacts. *Procedure* averages procedural coverage and state/data handling. *Constraints* measures whether strict task rules are preserved. *Safety* measures artifact hygiene, including conciseness and avoidance of brittle task-specific leakage or risky commands.

Figure 5 and Table 3 show qualitatively different artifacts. SKILLNET has the strongest grouped static score, driven by Environment and Grounding. SKILLCREATOR is strongest on Contract, Procedure, and Constraints. SKILLSEEKERS, despite the strongest Code and Doc averages in Appendix Table 6, has the best Safety score and strong Grounding but weaker Contract, Procedure, and Constraints.

Human validation on 100 generated skills, each independently rated by three annotators blind to method, shows high agreement on binary acceptance (Fleiss' $\kappa = 0.732$). Static scores correlate moderately with human Overall ratings (Spearman's $\rho = 0.451$, $p = 2.5 \times 10^{-6}$). Together with the differing static and execution rankings, these results support using the diagnostics as coarse artifact-quality screens alongside execution-based pass@3. Appendix A.5 reports the annotation protocol and diagnostic errors.

4.4 ERROR ANALYSIS

We analyze completed verifier failures: trials in which the executor invokes a generated skill and produces an answer rejected by the verifier. Five graduate students completed failure attribution using execution traces, generated code, and task metadata.

Figure 6 reports failure composition under the MiniMax-2.5 configuration. Code Repo failures are dominated by runtime or dependency issues (1245, 53%), followed by interface or schema errors (626, 27%) and asset or artifact issues (475, 20%). Code Doc failures are more concentrated: interface or schema errors account for 450 failures (85%), with a smaller runtime or dependency bucket (61, 11%). Domain Knowledge Doc failures exhibit a different profile, with state or rule errors (369, 44%)

and numeric or formula errors (306, 37%) dominating, and fewer interface or schema errors (129, 15%).

These distributions connect source characteristics to execution failures. Repository tasks require operational details about environments, assets, and interfaces; code-documentation tasks emphasize schema and format precision; domain-document tasks require exact numeric, state, and rule encoding. Static coverage of these components helps diagnose errors, while execution determines whether the resulting procedures satisfy the task constraints.

5 CONCLUSION

We introduced SkillGenBench, a 211-task benchmark that separates skill generation from downstream execution to compare pipelines across repository and document sources. Performance varies with the generation method, backbone, and source material. Recovering implicit procedures from repositories remains particularly challenging.

Generated skills can contain the expected structural components yet fail on interface alignment, state handling, and procedural constraints. Static diagnostics help assess artifact quality, while execution-based pass@3 measures whether generated skills support successful task completion.

AI USE STATEMENT

We used large language models for benchmark task construction and for the skill-generation and execution pipelines described in Sections 3 and 4. We also used Codex to assist with experimental design, code implementation and debugging, result analysis and interpretation, manuscript drafting and revision, and figure preparation. Implementation checks and instance-specific verifiers were used to assess code changes and execution outputs. Human task review, static-skill annotation, and failure analysis complement these automated checks. The authors take responsibility for the final text, claims, results, and artifacts.

ETHICS STATEMENT

SkillGenBench evaluates software and document-based workflows in isolated benchmark environments. Human review focuses on task validity and failure attribution. Reuse of third-party repositories, documents, and media should respect their licenses, attribution requirements, and privacy constraints. Generated skills can inherit incorrect procedures or unsafe operations from source materials and models. They are research artifacts, and deployment requires security review beyond task-level benchmark verification.

REPRODUCIBILITY STATEMENT

Section 3 and Appendix F describe task construction, verification, and prompts. Section 4.1 and Appendices B and C specify model choices, runtimes, method adaptations, and evaluation budgets. Appendix A.1 records aggregation and coverage, with configuration-specific and cross-executor results in Appendices A.2 and A.4. Appendix A.5 describes the human validation of static diagnostics. The source package includes machine-readable counts and plotting scripts for the task taxonomy, main-result heatmap, and failure analysis.

REFERENCES

Salaheddin Alzubi, Noah Provenzano, Jaydon Bingham, Weiyuan Chen, and Tu Vu. Evoskill: Automated skill discovery for multi-agent systems. *arXiv preprint arXiv:2603.02766*, 2026.

Anthropic. Introducing the model context protocol. `https://www.anthropic.com/news/model-context-protocol`, 2024.

Anthropic. Equipping agents for the real world with agent skills. `https://www.anthropic.com/engineering/equipping-agents-for-the-real-world-with-agent-skills`, 2025. Anthropic Engineering Blog.

Anthropic. Claude code. `https://code.claude.com/docs/en/overview`, 2025a. Official documentation.

Anthropic. What's new in claude 4.5. `https://docs.claude.com/en/docs/about-claude/models/whats-new-sonnet-4-5`, 2025b. Official model documentation.

Anthropic. skill-creator. `https://github.com/anthropics/skills/tree/main/skills/skill-creator`, 2026. Anthropic Agent Skills repository.

Shuangrui Ding, Xuanlang Dai, Long Xing, Shengyuan Ding, Ziyu Liu, Yang JingYi, Penghui Yang, Zhixiong Zhang, Xilin Wei, Xinyu Fang, et al. Wildclawbench: A benchmark for real-world, long-horizon agent evaluation. *arXiv preprint arXiv:2605.10912*, 2026.

Shihan Dou, Ming Zhang, Zhangyue Yin, Chenhao Huang, Yujiong Shen, Junzhe Wang, Jiayi Chen, Yuchen Ni, Junjie Ye, Cheng Zhang, et al. Cl-bench: A benchmark for context learning. *arXiv preprint arXiv:2602.03587*, 2026.

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? *arXiv preprint arXiv:2603.15401*, 2026.

Xu Huang, Junwu Chen, Yuxing Fei, Zhuohan Li, Philippe Schwaller, and Gerbrand Ceder. Cascade: Cumulative agentic skill creation through autonomous development and evolution. *arXiv preprint arXiv:2512.23880*, 2025.

Yanna Jiang, Delong Li, Haiyu Deng, Baihe Ma, Xu Wang, Qin Wang, and Guangsheng Yu. Sok: Agentic skills–beyond tool use in llm agents. *arXiv preprint arXiv:2602.20867*, 2026.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *The twelfth international conference on learning representations*, 2023.

Yusuf Karaaslan. Skill seekers. https://github.com/yusufkaraaslan/Skill_Seekers, 2026. Repository for converting documentation websites, GitHub repositories, and PDFs into Claude-compatible skills.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33:9459–9474, 2020.

Hao Li, Chunjiang Mu, Jianhao Chen, Siyue Ren, Zhiyao Cui, Yiqun Zhang, Lei Bai, and Shuyue Hu. Organizing, orchestrating, and benchmarking agent skills at ecosystem scale. *arXiv preprint arXiv:2603.02176*, 2026a.

Xiangyi Li, Wenbo Chen, Yimin Liu, Shenghan Zheng, Xiaokun Chen, Yifeng He, Yubo Li, Bingran You, Haotian Shen, Jiankai Sun, et al. Skillsbench: Benchmarking how well agent skills work across diverse tasks. *arXiv preprint arXiv:2602.12670*, 2026b.

Yuan Liang, Ruobin Zhong, Haoming Xu, Chen Jiang, Yi Zhong, Runnan Fang, Jia-Chen Gu, Shumin Deng, Yunzhi Yao, Mengru Wang, et al. Skillnet: Create, evaluate, and connect ai skills. *arXiv preprint arXiv:2603.04448*, 2026.

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 *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=zAdUB0aCTQ.

Yujian Liu, Jiabao Ji, Li An, Tommi Jaakkola, Yang Zhang, and Shiyu Chang. How well do agentic skills work in the wild: Benchmarking llm skill usage in realistic settings. *arXiv preprint arXiv:2604.04323*, 2026.

Zijian Lu, Yiping Zuo, Yupeng Nie, Xin He, Weibei Fan, Lianyong Qi, and Shi Jin. Contractskill: Repairable contract-based skills for multimodal web agents, 2026. URL https://arxiv.org/abs/2603.20340.

Ziyu Ma, Shidong Yang, Yuxiang Ji, Xucong Wang, Yong Wang, Yiming Hu, Tongwen Huang, and Xiangxiang Chu. Skillclaw: Let skills evolve collectively with agentic evolver. *arXiv preprint arXiv:2604.08377*, 2026.

Mike A Merrill, Alexander G Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026.

MiniMax. Minimax m2.5. https://www.minimax.io/news/minimax-m25, 2026a.

MiniMax. MiniMax M2.7. https://www.minimax.io/news/minimax-m27-en, 2026b.

Jingwei Ni, Yihao Liu, Xinpeng Liu, Yutao Sun, Mengyu Zhou, Pengyu Cheng, Dexin Wang, Erchao Zhao, Xiaoxi Jiang, and Guanjun Jiang. Trace2skill: Distill trajectory-local lessons into transferable agent skills, 2026. URL https://arxiv.org/abs/2603.25158.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, dahai li, Zhiyuan Liu, and Maosong Sun. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=dHng2O0Jjr.

Qwen Team. Qwen3.6-Plus: Towards real world agents, April 2026. URL https://qwen.ai/blog?id=qwen3.6.

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. *Advances in neural information processing systems*, 36:68539–68551, 2023.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.

Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, et al. Openai gpt-5 system card. *arXiv preprint arXiv:2601.03267*, 2025.

Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, SH Cai, Yuan Cao, Y Charles, HS Che, Cheng Chen, Guanduo Chen, et al. Kimi k2. 5: Visual agentic intelligence. *arXiv preprint arXiv:2602.02276*, 2026.

Chenxi Wang, Zhuoyun Yu, Xin Xie, Wuguannan Yao, Runnan Fang, Shuofei Qiao, Kexin Cao, Guozhou Zheng, Xiang Qi, Peng Zhang, et al. Skillx: Automatically constructing skill knowledge bases for agents. *arXiv preprint arXiv:2604.04804*, 2026.

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. *arXiv preprint arXiv:2305.16291*, 2023.

Peng Xia, Jianwen Chen, Hanyang Wang, Jiaqi Liu, Kaide Zeng, Yu Wang, Siwei Han, Yiyang Zhou, Xujiang Zhao, Haifeng Chen, Zeyu Zheng, Cihang Xie, and Huaxiu Yao. SkillRL: Evolving agents via recursive skill-augmented reinforcement learning. In *ICLR 2026 Workshop on Memory for LLM-Based Agentic Systems*, 2026. URL https://openreview.net/forum?id=By7Pj576U3.

Yutao Yang, Junsong Li, Qianjun Pan, Bihao Zhan, Yuxuan Cai, Lin Du, Jie Zhou, Kai Chen, Qin Chen, Xin Li, et al. Autoskill: Experience-driven lifelong learning via skill self-evolution. *arXiv preprint arXiv:2603.01145*, 2026.

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 *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Aohan Zeng, Xin Lv, Zhenyu Hou, Zhengxiao Du, Qinkai Zheng, Bin Chen, Da Yin, Chendi Ge, Chenghua Huang, Chengxing Xie, et al. Glm-5: from vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*, 2026.

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

Boyuan Zheng, Michael Y Fatemi, Xiaolong Jin, Zora Zhiruo Wang, Apurva Gandhi, Yueqi Song, Yu Gu, Jayanth Srinivasa, Gaowen Liu, Graham Neubig, et al. Skillweaver: Web agents can self-improve by discovering and honing skills. *arXiv preprint arXiv:2504.07079*, 2025.

YanZhao Zheng, ZhenTao Zhang, Chao Ma, YuanQiang Yu, JiHuan Zhu, Baohua Dong, and Hangcheng Zhu. Skillrouter: Retrieve-and-rerank skill selection for llm agents at scale. *arXiv preprint arXiv:2603.22455*, 2026.

Huichi Zhou, Siyuan Guo, Anjie Liu, Zhongwei Yu, Ziqin Gong, Bowen Zhao, Zhixun Chen, Menglong Zhang, Yihang Chen, Jinsong Li, et al. Memento-skills: Let agents design agents. *arXiv preprint arXiv:2603.18743*, 2026.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. Webarena: A realistic web environment for building autonomous agents. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=oKn9c6ytLx.

A Additional Results

A.1 Main-Result Aggregation and Coverage

The benchmark inventory contains 211 unique task IDs. For Table 2, the MiniMax-2.5 configuration covers 123 repository, 28 code-documentation, and 36 domain-documentation tasks with a 1800-second per-instance evaluation limit. The GPT-5.5 configuration covers 12 structured-domain tasks, four media-API tasks, four media-workflow tasks, and four native-vision tasks, with 600 seconds including queue time and 40 turns per trial, and at most three trials per task. These task-ID sets are disjoint.

For method m and generation backbone b, Table 2 reports $100 \sum_g S_{m,b,g} / \sum_g E_{m,b,g}$, where S counts successful tasks and E counts evaluable tasks in configuration g. This is a descriptive aggregate across configurations with different task coverage; method comparisons use matched task sets. Table 4 summarizes the resulting pass@3 scores; exact success and evaluation counts are included in the accompanying source data.

MiniMax-2.5 results are available as source-level summaries: integer success counts are uniquely recovered from the reported one-decimal rates and source denominators. GPT-5.5 results use task-level records. Task-level paired confidence intervals for the combined matrix cannot be recomputed from these summaries.

The GPT-5.5 generation matrix produced 397 valid source packages out of 475 attempts on available models; 78 attempts did not produce valid packages. Kimi K2.5 was unavailable for the remaining 95 scheduled source units. These outcomes yield 492 evaluated skill–task combinations, with 366 successes; 108 combinations have no valid package and 120 have no available model. Missing executions are excluded, not converted into failed trials. Kimi results from the MiniMax-2.5 configuration remain in the aggregate. The 24 workflow, reasoning, and visual task specifications and verifiers have completed dual human review.

Table 4: Descriptive pass@3 (%) across evaluated task configurations, corresponding to Table 2. Task coverage varies across cells.

Method	Sonnet 4.5	GPT-5	Kimi K2.5	GLM-5	MiniMax M2.7	Qwen3.6 Plus
Naive Prompt	19.0	21.8	12.3	24.9	21.8	24.2
EvoSkill	20.7	25.2	10.2	14.3	14.9	22.1
SkillNet	17.5	25.0	12.8	20.6	14.6	25.6
SkillCreator	17.0	23.2	12.8	27.0	18.5	24.6
SkillSeekers	24.6	28.9	14.4	26.1	24.2	26.1

Controls. No-skill execution succeeds on 44 of the 211 tasks (20.9%). Table 5 reports the same shared controls restricted to each skill cell's evaluated task set. Under the GPT-5.5 configuration, the shared no-skill and full-source controls solve 12 and 22 of 24 tasks. Full-source coverage is limited to this configuration.

A.2 Configuration-Specific Results

Table 6 reports the repository/documentation breakdown under MiniMax-2.5, and Table 7 gives successes and evaluable task counts under GPT-5.5. Within-generator intersections across the five methods contain 2, 20, 2, 12, and 12 GPT-5.5 tasks for Sonnet, GPT-5, GLM-5, MiniMax, and Qwen, respectively. The intersection across all five available generators and all methods is empty; comparisons therefore use the relevant matched task sets rather than assuming a common full matrix.

A.3 Method–Backbone Heatmap

Figure 7 visualizes the same descriptive aggregate as Table 2; it uses identical success counts and denominators. Coverage and execution configurations are specified above.

Table 5: No-skill pass@3 (%) on tasks matched to each method–backbone cell of Table 2. These are restrictions of shared controls, not independent baseline runs.

Method	Sonnet 4.5	GPT-5	Kimi K2.5	GLM-5	MiniMax M2.7	Qwen3.6 Plus
NAIVE PROMPT	20.5	20.9	17.1	21.1	20.9	20.9
EVOSKILL	20.7	21.0	17.1	18.0	18.8	18.6
SKILLNET	18.6	21.2	17.1	20.1	20.9	20.9
SKILLCREATOR	18.6	20.9	17.1	20.9	20.9	20.9
SKILLSEEKERS	19.8	20.9	17.1	20.9	20.9	20.9

Table 6: Source-stratified pass@3 (%) under the MiniMax-2.5 configuration. Code contains 123 repository tasks; Doc contains 28 code-documentation and 36 domain-documentation tasks. Avg. is the mean over six generation backbones.

Method	Sonnet 4.5		GPT-5		Kimi K2.5		GLM-5		MiniMax M2.7		Qwen3.6 Plus		Avg.	
	Code	Doc	Code	Doc	Code	Doc	Code	Doc	Code	Doc	Code	Doc	Code	Doc
NO SKILL	13.8	23.4	13.8	23.4	**13.8**	**23.4**	13.8	23.4	**13.8**	23.4	13.8	23.4	13.8	23.4
NAIVE PROMPT	7.3	23.4	12.2	21.9	9.8	17.2	**16.3**	23.4	13.0	20.3	14.6	25.0	12.2	21.9
EVOSKILL	9.8	**26.6**	14.6	**31.2**	7.3	15.6	11.4	17.2	6.5	20.3	15.4	**28.1**	10.8	23.2
SKILLNET	11.4	23.4	**17.9**	21.9	9.8	18.8	13.8	21.9	5.7	14.1	**16.3**	**28.1**	12.5	21.4
SKILLCREATOR	10.6	25.0	14.6	23.4	8.9	20.3	**16.3**	26.6	7.3	18.8	15.4	23.4	12.2	22.9
SKILLSEEKERS	**16.3**	25.0	17.1	28.1	9.8	**23.4**	14.6	**28.1**	13.0	**21.9**	15.4	23.4	**14.4**	**25.0**

A.4 CROSS-EXECUTOR EVALUATION

We compare GPT-5.5 and GLM-5.3 on the 20 tasks in the benchmark inventory with completed runs for both executors under all four conditions: 12 structured-domain tasks, four media-API tasks, and four media-workflow tasks. The intersection was defined after execution by benchmark membership and coverage, independently of task success. The four native-vision tasks have GPT-5.5 results only because the GLM endpoint did not pass the image-input qualification check.

GPT-5.5 generates the task-agnostic Naive Prompt and source-only two-stage EvoSkill packages. Each task uses the same frozen package, inputs, and verifier under both executors. No Skill and Full Source provide controls without generated skills and with direct source access, respectively. These controls belong to the paired executor runs; Table 2 uses its configuration-specific controls. All trials allow 600 seconds including queue time and 40 turns, with at most three attempts and early stopping after success. All 160 task–executor–condition combinations have execution outcomes.

Against No Skill, Naive Prompt gains five tasks and loses one under GPT-5.5, versus six gains and two losses under GLM-5.3. EvoSkill gains two and loses none under GPT-5.5, versus three gains and

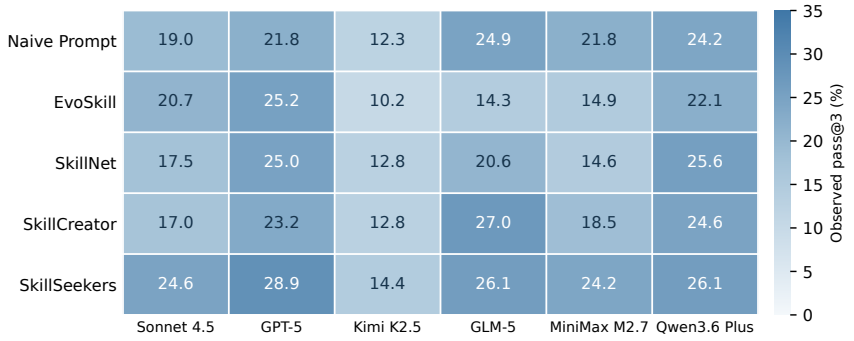


Figure 7: Descriptive observed pass@3 (%) corresponding to Table 2. Task coverage varies across cells.

Table 7: Successes / evaluated tasks under the GPT-5.5 configuration. Each method–backbone condition has 24 scheduled tasks. N/A denotes the unavailable Kimi K2.5 endpoint.

Method	Sonnet 4.5	GPT-5	Kimi K2.5	GLM-5	MiniMax M2.7	Qwen3.6 Plus
NAIVE PROMPT	16/23	17/24	N/A	17/22	17/24	17/24
EVOSKILL	14/21	15/23	N/A	2/2	9/15	7/12
SKILLNET	5/7	16/21	N/A	10/12	14/19	16/24
SKILLCREATOR	4/7	16/24	N/A	20/24	18/24	18/24
SKILLSEEKERS	15/20	22/24	N/A	19/24	21/24	21/24

Table 8: Cross-executor pass@3 on 20 shared benchmark tasks. Entries are successes / tasks; aggregate rates are shown in parentheses. The generator is fixed to GPT-5.5.

Task group	Executor	No Skill	Full Source	Naive	EvoSkill
All tasks	GPT-5.5	11/20 (55%)	17/20 (85%)	15/20 (75%)	13/20 (65%)
	GLM-5.3	11/20 (55%)	17/20 (85%)	15/20 (75%)	12/20 (60%)
Structured domain	GPT-5.5	5/12	11/12	9/12	7/12
	GLM-5.3	5/12	11/12	9/12	6/12
Media API	GPT-5.5	4/4	4/4	4/4	4/4
	GLM-5.3	4/4	4/4	4/4	4/4
Media workflow	GPT-5.5	2/4	2/4	2/4	2/4
	GLM-5.3	2/4	2/4	2/4	2/4

two losses under GLM-5.3. Thus, similar aggregate scores can conceal different task-level outcomes. Every condition achieves 4/4 on media-API tasks and 2/4 on media-workflow tasks, so the aggregate skill gains arise from the structured-domain group.

Pass@3 includes provider refusals, budget exhaustion, and skill-invocation failures, including image-capability refusals within media workflows.

A.5 HUMAN VALIDATION OF STATIC DIAGNOSTICS

We sampled 100 generated skills and asked three annotators to independently rate each skill on the six axes in Figure 5, using a 1–5 scale, and cast a binary Accept vote. Annotators were blind to the generation method. All three agreed on the Accept decision for 80 of the 100 skills, with Fleiss' $\kappa = 0.732$; pairwise Spearman correlations on Overall ratings ranged from 0.89 to 0.96. The static quality score correlated moderately with human Overall ratings (Spearman's $\rho = 0.451$, $p = 2.5 \times 10^{-6}$) and weakly discriminated majority-vote acceptance (AUC $= 0.604$).

The static averages use recursively collected `SKILL.md` file records, including nested files, with coverage varying by method and backbone. The eight checks were derived from recurring structural patterns and failure modes and grouped into six axes. Scoring applies deterministic rules without model calls. The diagnostics identified gross artifact defects, including an irrelevant placeholder skill rejected by all three annotators. Two subtler defects, a missing execution threshold and a skill consisting mainly of a function index, were detected by human judgment but missed by the static rules. These observations support the diagnostics as reproducible quality screens, with execution-based pass@3 providing the primary measure of downstream utility.

B MODEL AND HARNESS CONFIGURATIONS

Runtime harness. All experiments use the same SkillGenBench harness and Claude Code runtime. The main generation matrix uses the six backbones in Section 4.1, with downstream executors mapped in Appendix A.1. The paired executor comparison in Appendix A.4 fixes the generator to GPT-5.5 and varies the executor between GPT-5.5 and GLM-5.3. We run agent interactions through Claude Code CLI 2.1.85 with `claude-agent-sdk` 0.1.64.

Execution environment. Downstream evaluations are executed in isolated Docker environments selected by the instance configuration. No GPU resources are requested.

MiniMax-2.5 task-configuration generation parameters.

- Temperature: 0
- Max output tokens: 16,384
- Max rounds: 3 refinement iterations / 45 agent turns
- Timeout: 1800 seconds

MiniMax-2.5 evaluation parameters. The MiniMax-2.5 task configuration uses the following executor parameters:

- Temperature: 0
- Max output tokens: 16,384
- Timeout: 1800 seconds

GPT-5.5 task configuration. Source-only generators receive up to 120,000 characters and have no access to task instructions, inputs, answers, tests, or evaluation feedback. Naive Prompt, SkillNet, SkillCreator v2, and SkillSeekers use a 600-second generation-stage budget and a 16,384-token output ceiling. EvoSkill uses source-only proposer/generator stages with 12/20-turn limits and 600 seconds per stage; it does not consume evaluation trajectories in this configuration. SkillCreator v2 uses its authoring guide, and SkillSeekers uses the pinned 3.5.1 local-source wrapper. The evaluation limits are listed in Appendix A.1. The main matrix uses source-only generation for this configuration.

Generation-regime comparison. Figure 4 fixes the generator and executor to GPT-5.5 and compares two information settings on 24 tasks drawn from 19 sources. The conditioned generator receives source materials and the public instruction; task inputs, images, answers, tests, and evaluation feedback remain hidden. EvoSkill uses a spec-only proposer/generator adaptation with 12/20 turns and 600 seconds per stage. One conditioned EvoSkill generation reaches the 20-turn limit, leaving 119 valid packages and 23 tasks evaluable for all five methods under both regimes. The plot uses this common task set, selected by package availability. Evaluation permits 600 seconds including queue time, 40 turns, and up to three trials with success-based early stopping. Task-agnostic outcomes come from nonconcurrent runs matched on task inputs, executors, API routes, and budgets; all gains and regressions are retained. Method-matched source-cluster intervals include zero. These regime-comparison runs are separate from Table 2.

C Baseline Methods

We compare five skill-generation baselines in three families: prompt-based generation, workflow-based generation, and self-evolving generation. All baselines output a `SKILL.md` skill package. Execution settings are fixed within each task configuration; generation inputs and method adaptations are specified below.

Prompt-based skill generation. Naive Prompt is the minimal prompt-based baseline. The generator receives the visible corpus and, in the task-conditioned setting, the task instruction, then directly writes a skill package in a single generation pass. This baseline measures how much reusable procedural knowledge can be distilled from the exposed materials without trajectories, search, self-evaluation, or an explicit skill-authoring workflow.

Workflow-based skill generation. We include three system-level baselines that construct skills through explicit workflows. SkillNet Liang et al. (2026) represents toolkit-mediated skill creation, where source materials are transformed through a dedicated skill creation interface. SkillSeekers Karaaslan (2026) represents source-to-skill conversion pipelines for repositories and documents, emphasizing the extraction and packaging of actionable knowledge from external materials. SkillCreator Anthropic (2026) represents iterative skill authoring, where an agent drafts, evaluates, and

refines a skill before submission. These baselines share the same final interface, but differ in the inductive bias imposed by the construction process: toolkit-based packaging, source conversion, and self-refined authoring.

Self-evolving skill generation. EvoSkill Alzubi et al. (2026) represents methods that derive skills from execution experience rather than static context alone. In the MiniMax-2.5 task configuration, EvoSkill receives the visible corpus together with trajectories collected from corresponding runs without generated skills. This setting tests whether observed execution behavior provides useful procedural evidence for skill generation while preserving the benchmark boundary: trajectories are generated from the same visible task environment, and hidden tests or verifier internals are never exposed.

Benchmark integration. We format generator inputs, route model calls, and normalize the `SKILL.md` layout. Source-only generation receives collection-level materials; public-instruction-conditioned generation also receives the task specification. Under GPT-5.5 execution, EvoSkill uses the source-only two-stage adaptation and SkillCreator v2 uses its authoring guide. The MiniMax-2.5 configuration uses EvoSkill proposer–generator assets with benchmark-collected no-skill trajectories. This trajectory-based adaptation differs from both source-only generation and EvoSkill's native multi-round self-improvement loop.

D Case Studies

The following cases show abbreviated public instructions, relevant procedural knowledge, and reference outputs.

Case 1: StyleTransfer_gtb01: Neural Style Transfer (Image)

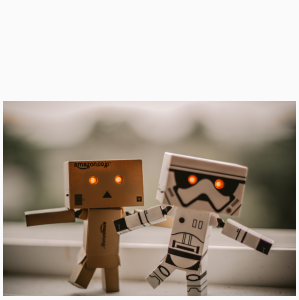
Task

Apply the style of a given artwork to a content photograph using the StyleTransfer repository, and save the stylized output as `styled_image.jpg`.

Skill-Dependent Knowledge

The generated skill must capture the repo's command-line interface, default optimization parameters, and output naming conventions. Without the skill, a generic agent may invoke the wrong entry point, use incorrect argument names, or save the output to an unexpected location.

Inputs and Reference Output



(a) Content image



(b) Style image



(c) Stylized output

The stylized result transfers the vibrant swirling patterns and warm color palette of the style painting onto the content photograph, while preserving the overall structure and recognizable shapes of the robot figurines.

Case 2: AnimeGANv3_gen04: Anime Style Transformation (Image)

Task

Transform a mountain landscape photograph into Shinkai Makoto animation style using the AnimeGANv3 library, preserving the original 2048×1365 pixel dimensions. Save the result as a properly color-corrected PNG file.

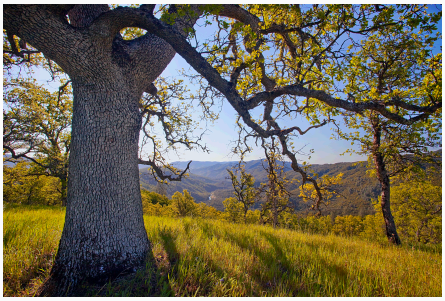
Skill-Dependent Knowledge

AnimeGANv3 uses BGR color channel ordering internally, diverging from the standard RGB convention. The repo provides explicit color space conversion utilities that must be applied during pre- and post-processing. An agent unfamiliar with this requirement will produce images with swapped color channels (warm tones appear cool and vice versa). Additionally, the agent must select the correct Shinkai-specific ONNX model variant from among multiple style options.

Inputs and Reference Output



(a) Input photograph



(b) Shinkai-style output

The output exhibits characteristic Shinkai-style atmospheric effects, enhanced color grading, and stylized lighting while maintaining the original image dimensions.

Case 3: Faker_gen08: Synthetic User Profile Generation (Code)

Task

Using the Faker library, generate a JSON file containing exactly 100 synthetic user profiles for an e-commerce platform. Each profile must include `username`, `email`, `ipv4`, and `user_agent` fields. All usernames must be unique. The output must be deterministically reproducible under `Faker.seed(24680)`.

Skill-Dependent Knowledge

1. The reference procedure uses Faker's `fake.unique.user_name()` proxy to enforce username uniqueness; plain `fake.user_name()` calls can produce duplicates.

2. The correct snake_case method names (`user_name`, `ipv4`) differ from common alternatives (`userName`, `ipv4_address`) that would raise errors or produce different outputs.

3. Class-level seeding via `Faker.seed(24680)` precedes generation to make the generated profiles reproducible.

4. The full public instruction specifies `json.dumps(..., indent=2, ensure_ascii=False)` and a trailing newline. The verifier parses and normalizes JSON, accepting equivalent serializations.

Expected Output (excerpt)

```
[
  {
    "username": "sarah34",
    "email": "michael45@example.com",
    "ipv4": "192.168.1.42",
    "user_agent": "Mozilla/5.0 ..."
  },
  ...  // 100 objects total, all usernames unique
]
```

Case 4: PDFPlumber_gen03: PDF Text Extraction Statistics (Code)

Task

Using the PDFPlumber library, analyze a technical manual PDF and produce a JSON file listing, for each page, the `page_number`, `word_count`, and `line_count`.

Skill-Dependent Knowledge

1. Word counting must use `page.extract_words()`, which returns word-level bounding box objects. Naive whitespace splitting of extracted text produces different counts around punctuation and ligatures.

2. Line counting must use `page.extract_text()` with the default `layout=False` setting and split on newline characters. Using `layout=True` or inferring lines from word y-coordinates yields different counts.

3. The JSON output records `page_number`, `word_count`, and `line_count` in page order. Object-key ordering does not change the parsed values.

Expected Output (excerpt)

```
[
  {"page_number": 1, "word_count": 312, "line_count": 28},
  {"page_number": 2, "word_count": 445, "line_count": 35},
  ...
]
```

Without the skill, agents commonly miscount words by splitting text on whitespace or miscount lines by using layout-preserved extraction, producing structurally valid but numerically incorrect outputs.

E LIMITATIONS

SkillGenBench evaluates skill generation under fixed execution budgets and task verifiers. It does not cover interactive negotiation with users, open-ended external-service workflows, or skill revision after deployment. Larger repositories, multi-repository workflows, and longer-lived skill libraries would test transfer beyond the current sources.

Failure categories summarize mechanisms that can overlap within a trial. Static diagnostics characterize artifact structure and can miss finer procedural defects, as the human validation in Appendix A.5 shows. Aggregation and coverage limits are specified in Appendix A.1.

F BENCHMARK CONSTRUCTION DETAILS

F.1 HUMAN VERIFICATION

After automated screening, reviewers examine each candidate's source materials, task specification, tests, verifier, and exposed skill materials against the five criteria in Table 9.

The final benchmark contains 211 human-reviewed tasks. Candidates with repairable issues were revised and rechecked; other candidates were discarded.

Verifier invariance. In the MiniMax-2.5 configuration, a metamorphic audit re-serialized JSON, re-quoted CSV, and re-spaced text for all 21 eligible tasks; all 21 preserved their verification outcome. JSON exact-mode checks parse and normalize content before comparison, so indentation, ASCII escapes, and object-key ordering do not determine success.

F.2 GENERATION PROMPTS

This section lists the prompts driving each stage of our generation pipeline. We follow Python's `str.format` convention: single braces `{var}` mark runtime substitutions (e.g., the source document, the KG summary, the slot index), while doubled braces `{{...}}` are literal braces forwarded to the model, which is used primarily inside the JSON schemas embedded in each prompt.

Table 9: Manual verification criteria for candidate benchmark tasks. A candidate is retained only when it satisfies all five criteria.

Criterion	Acceptance criterion	Typical failure mode
Moderate difficulty	Requires nontrivial use of the source materials while remaining solvable from the provided context	Too easy, unrealistic, ambiguous, or underspecified
Clear specification	States the expected inputs, outputs, constraints, and boundary conditions	Missing schema, vague target behavior, or conflicting requirements
Quantitative evaluability	Can be judged by executable tests, structured checks, or well-defined artifact metrics	Requires open-ended subjective judgment
Sufficient coverage	Test cases cover normal, edge, and adversarial conditions aligned with the task	Only covers the happy path or misses key constraints
Test-case quality	References, tests, and exposed materials are consistent and do not leak answers or verifier-specific hints	Answer leakage, brittle checks, or inconsistent expected outputs

F.2.1 Stage 1 — Knowledge Graph Construction

We extract a typed knowledge graph from each source document or code repository, then detect communities and summarize them into theme-level descriptors that serve as scenario-generation context downstream.

KG construction. A single-pass prompt that proposes entity types, extracts entities, and emits subject–predicate–object triples in one structured JSON object.

Prompt 1: KG Construction

```
## Task

Analyze the following text and perform three steps IN ORDER, producing
     a single
JSON output that contains the results of all three steps.

### Step 1: Propose Entity Types

Identify the entity types present in this text that are relevant to
    understanding
its content. The goal is to capture entities that have meaningful
    relationships
with other entities.

Rules:
- Avoid overly generic types like "other" or "unknown"
- Do NOT generate redundant or overlapping types (e.g., if the text
    has both
"company" and "organization", pick only one)
- Quality over quantity -- every type must be relevant

### Step 2: Extract Entities

Using the entity types you proposed in Step 1, extract all important
    entities
from the text.

Rules:
- Focus on substantive mentions -- entities central to the text or
    mentioned
with meaningful detail
```

```
- Prioritize entities with relationship potential -- they should
    connect to
other entities in the text
- Use the most widely recognized or official name as it appears in the
     text
- Use full names for people (e.g., "Marie Curie" not just "Curie")
- Normalize variants to one canonical form (e.g., "U.S." -> "United
    States")
- Each unique entity appears only once
- Be selective: a focused set of well-connected entities is better
    than a
comprehensive list with many isolates
- For each entity, provide:
  - id: a unique slug-style identifier (lowercase, underscores, e.g.,
    "bigfoot", "myth_track")
  - name: canonical name
  - type: one of the types proposed in Step 1
  - description: concise description (1-3 sentences) based on what the
     text says

### Step 3: Extract Relations

Extract subject-predicate-object triples between the entities
    identified in Step 2.

Rules:
- **Subject** and **object** must both be entities from Step 2 (use
    their IDs)
- **Predicate** should be clear and specific (e.g., "founded_by", "
    located_in",
"part_of", "interacts_with", "scored_by")
- Avoid vague predicates like "related_to" or "associated_with"
- Only extract relationships explicitly stated or clearly implied in
    the text
- Ensure correct directionality (subject -> predicate -> object)
- Be thorough: maximize connectivity, minimize isolated entities
- For each relation provide a brief source_text quote supporting it

## Output Format

Return a single JSON object with this exact structure:
```json
{{
  "entity_types": ["type1", "type2", ...],
  "entities": [
    {{
      "id": "entity_id",
      "name": "Entity Name",
      "type": "entity_type",
      "description": "What this entity is based on the text"
    }}
  ],
  "relations": [
    {{
      "subject_id": "entity_id_1",
      "predicate": "relationship_verb",
      "object_id": "entity_id_2",
      "source_text": "brief quote from text supporting this relation"
    }}
  ]
}}
```
```

```
## Text to Analyze

{text}

## Extraction

Analyze the text above and return the JSON output. Be thorough but
    precise.
```

Community summary. After running community detection on the merged KG, every community is condensed into a short thematic summary that anchors downstream scenario generation.

Prompt 2: Community Summary

```
## Task

Summarize the following group of related entities and relationships
    from a document in 2-3 sentences.

Focus on what topic or theme this group covers.

## Entities ({entity_count})

{entity_lines_joined}
```

F.2.2 Stage 2 — Scenario Generation

Conditioning on the KG summary together with the source document, we ask the model to propose practical, multi-section, computation-bearing application scenarios that motivate the downstream tasks.

Prompt 3: Scenario Generation

```
# Task Scenario Generation

You are a benchmark designer. Read the document below and propose **{
    num_scenarios}** realistic, practical application scenarios.

## Requirements

Each scenario must be a **real-world tool or system** that a human
    would actually build using this document's knowledge, such as:
- Rule engines, scoring systems, decision tools
- Automated assistants or advisory bots
- Process automation, workflow validators
- Data processing pipelines, calculators
- Compliance checkers, audit tools

### What makes a GOOD scenario:
- Requires integrating knowledge from **multiple sections** of the
    document
- Involves **computation, logic, or decision-making** -- not just
    information retrieval
- Would be genuinely useful to someone working in this domain
- Requires document-specific rules/data that cannot be guessed from
    common knowledge

### What to AVOID:
- Exam-style questions ("What is X?" or "List the Y")
- Simple fact lookups or single-step reasoning
- Scenarios where common knowledge suffices without the document
```

```
- Overly narrow scenarios that only touch one sentence in the document
- Scenarios that are too abstract or vague to implement

## Output Format

Output a JSON array of {num_scenarios} scenario objects:

```json
[
  {{
    "scenario_id": "scenario_001",
    "title": "Short descriptive title",
    "description": "2-3 sentences describing what the tool/system does
    ",
    "real_world_use_case": "Who would use this and why",
    "applicable_rules": ["Rule/section 1 from doc", "Rule/section 2"],
    "domain_concepts": ["concept1", "concept2"],
    "data_needed": ["What specific data from the document is needed"],
    "problem_type_suggestion": "MC|FIB|CS|RANK|CODE",
    "data_strategy_suggestion": "doc_lookup|doc_logic|
    synthetic_scenario",
    "difficulty_estimate": "medium|hard",
    "cross_section_refs": ["Section A", "Section B"]
  }}
]
```

Knowledge Graph Summary

{kg_summary}

Document

{document}
```

F.2.3 Stage 3 — Task and Test-Case Generation

For every scenario slot, this prompt jointly produces (i) a *function-interface* task description that abstracts away document-specific constants and (ii) an executable test-case bundle whose `solve` function hardcodes those constants internally, enforcing the contamination boundary central to our benchmark.

Prompt 4: Task / Test-Case Generation

```
# Function-Based Task & Test Case Generation

Generate ONE task and ONE testcase bundle for the given scenario.

## Scenario
{scenario}

## Task Slot: {slot_idx}

## Full Knowledge Graph
{kg_summary}

## KG Chain (focused subset for this task)
{kg_chain}

## KEY CONCEPT: Function-Based Tasks
```

```
Every task describes a **general tool/function interface** -- NOT a
    specific one-off question with fixed numbers.

- The QUESTION describes WHAT the function does, its INPUT schema, and
     its OUTPUT schema.
- The QUESTION references document rules in abstract terms (e.g., "
    according to the document's threshold", "as specified in the
    procedure").
- The TEST CASES provide {test_cases_per_task} **different concrete
    scenarios** as input->output pairs.
- The SOLUTION CODE defines a `solve(input_data)` function with
    document-specific constants hardcoded inside.

### WRONG (too specific -- all numbers baked in):
```
"question": "A scheduler validates two Tuesday-cycle threads. Thread A
     has walls [5,12,23] with ages [3,8,15] and threshold 10..."
```
This bakes ALL specifics into the question. Only one answer is
    possible, and test cases can only check variable names.

### CORRECT (function interface):
```
"question": "Implement `solve(input_data)` as a **Wall Scheduler**.\n\
    nINPUT: dict with 'threads' (list of thread dicts with 'walls', '
    max_age_threshold') and 'mandatory_walls' (list of IDs)\n\nOUTPUT:
     dict with 'selected_walls' per thread and 'halted' flag\n\nRULES
    :\n- Threshold is per-policy-excerpt (not global)\n- Noncompliance
     alert has selection precedence over age\n- Mandatory walls always
     included"
```
Test cases provide {test_cases_per_task} different inputs with
    different configurations and expected outputs.

## CRITICAL RULES

### Anti-Contamination
1. The **question text** describes the function interface using
    ABSTRACT references to document rules.
2. All **document-specific constants, thresholds, formulas, parameter
    values** are hardcoded INSIDE the `solve()` function body -- NOT
    in the question text, NOT in the test case inputs.
3. Test case inputs contain varying scenarios; the function applies
    document-internal knowledge to compute outputs.
4. The task must be IMPOSSIBLE to implement correctly without reading
    the document.

### Question Quality
5. Questions must require **multi-step reasoning** using multiple
    document rules.
6. Each question integrates information from **multiple parts** of the
     document.
7. The function should test **consequences and interactions** of
    document rules.
8. Input schema should be rich enough to support {test_cases_per_task}
     diverse test scenarios.

### Test Case Requirements
9. Generate exactly {test_cases_per_task} test cases with diverse
    scenarios.
10. Cover: normal cases, edge cases, boundary conditions, rule
    interactions.
```

```
11. Each test case: `{{"input": {{...}}, "expected_output": {{...}}}}`.

12. `solution_code` must define a function named `solve` taking one
    dict argument.
13. All document-specific values hardcoded inside `solve`.
14. `solve` must be self-contained -- no external variable references.
15. The `solve` function body should include comments citing which
    document section each constant comes from.

### Input Constant Consistency
16. All string constants in test case inputs (e.g., entity names, type
     names, category labels) MUST exactly match the strings used in `
    solve()` comparisons. If `solve()` checks `name == "Setting Manager
    "`, the test input must use `"Setting Manager"`, NOT `"
    SettingManager"` or `"setting_manager"`. Double-check: every
    string in input that will be compared inside `solve()` must be
    copy-pasted from the corresponding `solve()` code.

### Output Simplicity Constraints
17. The function output MUST be a FLAT or SHALLOW dict (max 2 nesting
    levels). No deeply nested reports, audits, or multi-section
    compilations.
18. Each test case expected_output MUST have at most 15 leaf values (
    strings, numbers, booleans, nulls). Count every terminal value in
    the nested structure. If your design exceeds 15, simplify the
    output schema.
19. The task MUST be a SINGLE DECISION or SINGLE COMPUTATION -- not "
    compile a full report". Good: "decide which walls to reset", "
    classify the input", "compute a score". Bad: "generate a
    comprehensive audit report", "compile a multi-section learning
    pack".
20. String values in expected_output SHOULD be short enums or codes (e
    .g., "PASS", "FAIL", "ALERT"), NOT long narrative sentences. If a
    reason/message field is needed, keep it under 50 characters.
21. The question text MUST be COMPLETE -- do not truncate input
    schemas or rule descriptions. If the schema is too large, simplify
     the function interface to fewer input fields. Every field
    mentioned in the INPUT section must have its type and description
    fully specified.

## Output Format

Output exactly TWO JSON blocks (```json fenced):

### Block 1: Task
```json
{{
  "task_id": "descriptive_tool_name_{slot_idx:03d}",
  "type": "tool_type",
  "question": "Implement a function `solve(input_data)` that acts as a
     **Tool Name**.\n\nINPUT: a dict with keys:\n- 'key1': type --
    description\n- 'key2': type -- description\n\nOUTPUT: a dict with
    keys:\n- 'result_key': type -- description\n\nRULES (from the
    document):\n- Rule 1 (abstract, no specific values)\n- Rule 2 (
    abstract, no specific values)",
  "expected_output": {{
    "format": "dict",
    "key_results": {{}}
  }},
  "info_locations": [{{"name": "...", "location": "...", "description
    ": "..."}}],
  "domain_knowledge_needed": [{{"knowledge": "...", "in_document":
    true}}],
```
```

```
  "reasoning_steps": ["step1", "step2"],
  "computation_chain_length": 5,
  "anti_contamination": {{
    "why_not_pretrain": "...",
    "why_skill_helps": "..."
  }}
}}
```

Block 2: Testcase Bundle
```json
{{
  "task_id": "same_as_above",
  "test_id": "tc_{slot_idx:03d}",
  "setup_code": "import math\nimport numpy as np",
  "function_name": "solve",
  "solution_code": "def solve(input_data):\n    # Extract input\n
    key1 = input_data['key1']\n    key2 = input_data['key2']\n    #
    Document-specific constants (from Section X.Y)\n    THRESHOLD = 42
      # Section 3.2\n    RATE = 0.15  # Table 1\n    # Apply document
    rules\n    ...\n    return {{'result_key': computed_value}}",
  "test_cases": [
    {{
      "input": {{"key1": "scenario_A_value", "key2": 10}},
      "expected_output": {{"result_key": "expected_A"}}
    }},
    {{
      "input": {{"key1": "scenario_B_value", "key2": 20}},
      "expected_output": {{"result_key": "expected_B"}}
    }},
    {{
      "input": {{"key1": "edge_case", "key2": 0}},
      "expected_output": {{"result_key": "expected_edge"}}
    }}
  ],
  "timeout_seconds": 120
}}
```

Difficulty Amplification
- The `solve()` function MUST hardcode at least 3 distinct document-
    specific constants from different sections.
- Rules MUST come from at least 2 different document topics/sections.
- Include at least 1 edge case where a naive interpretation gives a
    WRONG answer.
- Include at least 1 rule interaction that produces a counter-
    intuitive result.
- At least 2 of the {test_cases_per_task} test cases should trigger
    edge-case behavior.
- Constants should NOT be round numbers or common values -- use the
    exact values from the document.

Document

{document}
```

#### F.2.4 STAGE 4 — VALIDATION AND REFINEMENT

Each candidate task is screened along three orthogonal axes before acceptance: an LLM judge rates eight quality dimensions; a *corpus-free* solver attempt estimates pretrain-contamination risk; and a *with-corpus* solver attempt verifies that the document is actually sufficient. Failing tasks are routed back through a refinement prompt rather than discarded.
```

Multi-dimensional verification.

Prompt 5: Task Verification

```
# Multi-Dimensional Question Evaluation

Evaluate the quality of a benchmark question based on the evidence
    below.

## Evidence
{evidence}

## Evaluation Criteria

Rate each dimension on a 0.0-1.0 scale:

1. **complexity_score**: How many reasoning steps are required? Simple
     lookup = 0.2, multi-step computation = 0.8+
2. **utility_score**: How practical is this task? Would someone
    actually need to solve it? Exam-style = 0.2, real tool = 0.8+
3. **contamination_risk**: Based on pretrain_score, how likely is
    pretrain contamination? >20% pretrain = high risk
4. **doc_dependency**: How much does the answer depend on document-
    specific info? Generic knowledge suffices = 0.2, fully doc-
    dependent = 1.0
5. **skill_benefit**: How much would a good skill document help? No
    help = 0.2, critical = 1.0
6. **doc_only_sufficient**: Based on doc_only_score, can the task be
    solved with the document? <10% = critically too hard (needs
    simplification). 10-40% = ideal range. >50% = may be too easy.
7. **doc_adds_value**: Does the document provide advantage over
    pretrain alone? doc_only_score - pretrain_score < 10% = document
    not helping.
8. **output_testability**: Are the expected outputs structural and
    objectively verifiable?
   - Outputs dominated by long freeform strings (messages,
    recommendations, dialog >30 chars) test string reproduction rather
     than document knowledge -> score 0.0-0.2
   - Outputs that are numbers, booleans, short enums, computed values
    -> score 0.8+
   - Outputs with emoji characters -> score 0.0
   - Deeply nested outputs with >20 leaf values per test case -> score
     0.2 (too complex to match)
   - If >70% of test cases produce identical output -> score 0.2 (
    lacks discriminability)

## Also provide:
- **verdict**: "pass" or "fail"
- **fail_reasons**: list of specific reasons if failing
- **improvement_suggestions**: specific ways to make the question
    harder/better

## Output Format

```json
{{
  "complexity_score": 0.7,
  "utility_score": 0.8,
  "contamination_risk": "low|medium|high",
  "doc_dependency": 0.9,
  "skill_benefit": 0.7,
  "output_testability": 0.8,
  "verdict": "pass",
```
```

```
  "fail_reasons": [],
  "improvement_suggestions": ["suggestion 1"]
}}
```

Corpus-free solvability check. The candidate solver receives only the task statement; the resulting pass-rate estimates how much of the answer is recoverable from parametric knowledge alone (the contamination floor).

Prompt 6: Corpus-Free Solvability Check

```
Task: {question}

Write a Python function named `solve` that implements the tool
    described above.
The function takes a single dict argument and returns the output as
    described.

Requirements:
1. Import any necessary libraries at the top level
2. Define the `solve(input_data)` function
3. Hardcode all necessary constants inside the function
4. Return the result as specified in the task

Return ONLY executable Python code. No markdown fences, no
    explanations.
```

With-corpus triviality check. The same solver is rerun with the source document attached; doc_only − pretrain quantifies how much value the document actually contributes.

Prompt 7: With-Corpus Triviality Check

```
You are given the following reference document:

{document}

---

Task: {question}

Write a Python function named `solve` that implements the tool
    described above.
The function takes a single dict argument and returns the output as
    described.

Requirements:
1. Import any necessary libraries at the top level
2. Define the `solve(input_data)` function
3. Hardcode all necessary constants (from the document) inside the
    function
4. Return the result as specified in the task

Return ONLY executable Python code. No markdown fences, no
    explanations.
```

Targeted refinement. Rather than discarding rejected tasks, the verifier's failure reasons are forwarded to a refinement prompt that surgically fixes the identified issue (contamination, over-difficulty, string-matching output, low diversity, etc.) while preserving `task_id` and `test_id`.

Prompt 8: Task / Test-Case Refinement

```
# Question Improvement

The question below failed verification. Improve it based on the
    analysis.

## Original Task
{task}

## Original Testcase Bundle
{testcase}

## Verification Analysis
{analysis}

## Document Excerpt (for reference)
{document_excerpt}

## Improvement Instructions

Based on the failure analysis, improve the question and testcase:

1. **If pretrain contaminated**: Make the question more document-
    specific. The `solve` function should require document-specific
    constants/rules that cannot be guessed. Reference obscure details,
     combine multiple rules, or require document-specific parameter
    values hardcoded in the function body.
2. **If too simple**: Add more computation steps, require cross-
    referencing multiple sections, or ask about consequences rather
    than facts.
3. **If low utility**: Reframe as a practical tool or system that
    someone would actually build.
4. **If solution code broken**: Fix the `solve` function while keeping
     the question intent.
5. **If too hard (doc_only too low)**: The task is too complex for the
     LLM to solve even with the full document. Simplify by: reducing
    the number of rules/steps required, making the expected output
    format simpler (fewer nested keys), breaking the task into a
    smaller more focused scope, using more standard output types (
    single value, simple dict) instead of complex nested structures.
    Keep the core document-dependency but reduce implementation
    complexity.
6. **If doc_not_helping (doc_only =~ pretrain)**: The document doesn't
     provide meaningful advantage. Make the task more document-
    specific by: requiring document-specific constants that cannot be
    guessed, referencing obscure details or unique terminology from
    the document, combining rules from multiple non-obvious sections
    of the document.
7. **If string_matching_dominant**: The task's expected outputs rely
    on exact matching of long freeform strings (>30 chars). This tests
     string reproduction, NOT document knowledge. Restructure the
    output to test STRUCTURAL decisions:
   - Messages/replies -> action codes + rule IDs: `{{"reply": "Hold on
    !..."}}` -> `{{"action": "SAFETY_BLOCK", "rule_triggered": "
    R3_ELECTRICAL", "safety_level": "HIGH"}}`
   - Recommendations -> category enums + scores: `{{"advice": "Passage
     p1: paraphrase..."}}` -> `{{"risk_level": "HIGH", "primary_issue":
     "SIMILARITY_EXCEEDED", "exceeded_by": 1.0}}`
   - Status messages -> boolean flags + enums: `{{"message": "You are
    eligible!"}}` -> `{{"eligible": true, "action": "PROCEED", "
    redirect_target": null}}`
```

```
   - Descriptions -> structured facts: `{{"issue": "Absolute claim not
     allowed for 'Joker'..."}}` -> `{{"issue_type": "ABSOLUTE_CLAIM",
    "theme_keyword": "Joker", "violating_words": ["inherently"]}}`
   - All strings must be SHORT (<30 chars) from a CLOSED SET. NEVER
    use emoji. Add numeric/boolean outputs. Preserve the SAME document
     knowledge requirement.
8. **If output_overly_complex**: The task's expected output is too
    deeply nested (>20 leaf values per test case), causing format-
    sensitive failures even when the logic is correct. Simplify the
    output structure:
   - Flatten nested dicts: instead of `{{"responses": [{{"task_id":
    "...", "status": "...", "new_totals": {{"total_spent": ..., "
    by_category": {{...}}}}}}]}}`, return a simpler summary: `{{"
    total_spent": ..., "last_action_status": "success", "
    rejected_count": 0}}`
   - Reduce the number of output fields to the ESSENTIAL ones that
    test document knowledge
   - Remove intermediate/debugging fields (detailed per-command
    responses, full category breakdowns)
   - Keep at most 2 levels of nesting in the output dict
   - Focus on the FINAL computed result, not the step-by-step trace
9. **If emoji_in_output**: Remove ALL emoji characters from expected
    output values. Replace emoji-containing messages with structured
    action codes or boolean flags. Emoji make exact matching
    impossible for LLMs and do not test document knowledge.
10. **If low_output_diversity**: The task produces nearly identical
    output for most test cases, meaning it lacks discriminability --
    the LLM could score high just by returning a fixed template. Fix
    by:
    - Adding more varied input scenarios that trigger DIFFERENT code
    paths and produce DIFFERENT outputs
    - Ensuring at least 50% of test cases have structurally different
    expected outputs
    - Including edge cases and boundary conditions that produce
    distinct results
    - Each test case should exercise a different rule or combination
    of rules from the document

## CRITICAL RULES -- Function-Based Format
- Keep the same task_id and test_id
- The question describes a GENERAL function interface (input schema ->
     rules -> output schema)
- The question must NOT contain document-specific values/numbers
- `solution_code` must define a `solve(input_data)` function
- All document-specific constants hardcoded INSIDE the `solve`
    function
- Include 5-10 diverse test cases as input->output pairs
- Each test case tests a different scenario/edge case

## Output Format

Output exactly TWO JSON blocks (```json fenced): one improved task,
    one improved testcase bundle.
Use the same schema as the originals but with the function-based
    format.
```

F.3 Evaluation Prompts

Prompt 9: Agent Evaluation Instruction

```
Read /workspace/instruction.md for the full task description.
Input files are at /workspace/input/.
Documentation is at /workspace/docs/.
Write the complete Python solution to /workspace/agent_output.py. The
    file must be importable and define any required functions (e.g.
    solve()).
IMPORTANT: You MUST use the Skill tool at least once before
    implementing to load and follow the most relevant skill guidance.
If the triggered skill references bundled helper files such as scripts
    / or references/, use the mirrored skill package under /workspace/
    skill/.
```