WHEN SKILLS FAIL TO COMPOSE: BENCHMARKING AND COMPILING AGENT SKILLS

Anonymous authors
Paper under double-blind review

ABSTRACT

Agent skills provide reusable procedures and resources, but supplying relevant skills does not ensure that an agent can coordinate them to complete a shared task. We introduce **SkillsBench-Compose**, a benchmark of 20 human-authored tasks that evaluates component requirements and cross-component relationships separately. This separation exposes failures in integrating otherwise successful components. We also propose **SkillCompiler**, which jointly compiles the supplied skills into one task-specific package before execution. It generates a coordinated workflow grounded in critical source rules, records guidance and resource choices in a structured intermediate representation, and packages the guidance with selected original resources. Across three language models, SkillCompiler improves on the highest-scoring baseline in each setting by 2.9–4.8 reward points on SkillsBench and 3.3–8.3 percentage points in full completion on SkillsBench-Compose, while using fewer total model tokens, including preparation. Ablations show gains over task-only guidance and independently rewritten skills, including when the rewritten skills are consolidated into a single package.

1 INTRODUCTION

Agent skills package instructions, scripts, templates, and reference materials into reusable capabilities, allowing agents to draw on existing procedural knowledge for specialized tasks (Li et al., 2026b). Real tasks often require several skills to serve a shared goal. For example, building a searchable dialogue catalog requires parsing dialogue into a graph, creating a search interface, and linking each search hit to the correct graph node and story path. These operations must agree on the data they exchange and the objects they refer to.

Completing such a task requires both functioning components and correct relationships between them. Skills typically organize guidance around their own functions, while a shared task determines which upstream artifact a downstream step should consume, how different interfaces identify the same object, and how outputs jointly satisfy the final constraints. These task-specific relationships need not be specified within any individual skill. Even with high-quality, task-relevant skills already available, the agent must establish these connections and organize the skills into a coherent execution process. Recent work supports this process through runtime orchestration and preparation before execution: AgentSkillOS explicitly organizes skill dependencies and execution (Li et al., 2026a), while SkillRAE compiles task-specific context for selected skills (Meng et al., 2026). Evaluating skill use therefore requires examining whether agents satisfy the relationships between skills as well as carry out their individual procedures.

Overall gains from relevant skills can mask losses on multi-skill task subsets. We examine this variation on SkillsBench, whose task-associated skills undergo quality screening and human review (Li et al., 2026b). Building on its analysis by skill count, we compare three models on 87 tasks under No-skill, Raw (original packages), and Concat (concatenated instructions with resources retained).

Raw and Concat improve mean reward both overall and on the pooled multi-skill subset for all three models (Table 1). However, gains are consistently smaller in the four-or-more-skill group than in either lower-count group. In this group, Raw still helps GPT-5.4 Mini but lowers reward for the other two models; Concat lowers Mini's reward and yields near-zero gains for GPT-5.4 and Qwen3.7-plus. These task-level rewards reveal uneven skill utility, but leave the source of failure unresolved.

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.

Table 1: SkillsBench rewards (0–100). Subsets contain 23/43/21 tasks; Multi combines the latter two (64), and Overall includes all 87. Parentheses and blue/red shading show changes from No-skill. Single-skill Concat reuses Raw.

Method	1 skill	2–3	4+	Multi	Overall
GPT-5.4 Mini					
No-skill	42.03	27.44	30.48	28.44	32.03
Raw	56.52 (+14.49)	40.58 (+13.14)	37.62 (+7.14)	39.61 (+11.17)	44.08 (+12.05)
Concat	56.52 (+14.49)	40.58 (+13.14)	23.81 (−6.67)	35.08 (+6.64)	40.75 (+8.72)
GPT-5.4					
No-skill	33.33	39.07	49.84	42.60	40.15
Raw	56.52 (+23.19)	55.63 (+16.56)	43.65 (−6.19)	51.70 (+9.09)	52.97 (+12.82)
Concat	56.52 (+23.19)	55.12 (+16.05)	50.16 (+0.32)	53.49 (+10.89)	54.29 (+14.14)
Qwen3.7-plus					
No-skill	27.08	30.93	28.57	30.16	29.34
Raw	49.82 (+22.73)	50.12 (+19.19)	19.05 (−9.52)	39.92 (+9.77)	42.54 (+13.19)
Concat	49.82 (+22.73)	53.05 (+22.12)	29.05 (+0.48)	45.17 (+15.02)	46.40 (+17.06)

Separating errors within components from failures to connect otherwise successful components requires checking component requirements and cross-component relationships independently.

To make this distinction measurable, we introduce **SkillsBench-Compose** (§3), a benchmark of 20 human-authored tasks built from SkillsBench components. Each task retains the source skill packages and component acceptance requirements, and adds explicit integration checks. These checks test artifact handoffs, shared objects or state, and joint constraints on outputs. Full completion requires passing both the component and integration checks; separate component and integration metrics identify which requirements remain unmet. Every task was approved by three independent reviewers, and the benchmark was frozen before method evaluation.

These task-specific relationships also motivate **SkillCompiler** (§4), which prepares the supplied skills jointly before execution. An artifact handoff, for example, requires considering both the producing procedure and the downstream consumer. Inspired by program compilation, SkillCompiler encodes a coordinated workflow, critical source rules, and resource choices in a structured intermediate representation. A program validates its structure and references, then assembles the guidance with selected original resources into one task-specific skill package. The existing executor uses this package in place of the supplied package set, with guidance that connects the procedures and the resources needed to carry them out.

Across GPT-5.4 Mini, GPT-5.4, and Qwen3.7-plus, SkillCompiler achieves the highest overall SkillsBench reward and Compose full completion rate among the evaluated methods. Compose also reveals executions in which every component passes its checks but integration fails, demonstrating the need to evaluate cross-component relationships explicitly (§5.2). Ablations on multi-skill tasks help explain these gains: task-only guidance underperforms SkillCompiler, and consolidating the same independently rewritten skills into one package improves their performance but leaves a gap to joint compilation with every backbone (§5.3). Including preparation, SkillCompiler also uses fewer total model tokens than the highest-scoring baseline in each main comparison (§5.4).

We contribute a benchmark that exposes integration failures hidden by component-level success, and a compiler that coordinates existing skill packages for a shared task. Together, these contributions make skill composition directly measurable and demonstrate the benefits of joint preparation before execution.

2 Related Work

Agent benchmarks and compositional evaluation. Agent benchmarks evaluate multi-step interaction across environments (Liu et al., 2024; Zhou et al., 2024; Xie et al., 2024) and stateful application workflows (Trivedi et al., 2024; Yao et al., 2024). ToolBench and TaskBench examine API use

and dependencies between calls (Qin et al., 2024; Shen et al., 2024), while WorkArena++ composes atomic workplace tasks into larger workflows (Boisvert et al., 2024). SkillsBench and SWE-Skills-Bench measure the utility of procedural packages in executable tasks (Li et al., 2026b; Han et al., 2026). SkillsBench-Compose retains the source skill packages and component acceptance criteria, and evaluates cross-component relationships separately. This design exposes integration failures even when each component succeeds.

Reusable skills and experience. Reusable procedural knowledge can take the form of executable skills (Wang et al., 2023; Cai et al., 2024; Zheng et al., 2025), reflective feedback and experiential guidance (Shinn et al., 2023; Zhao et al., 2024), or workflows and skill directories distilled from trajectories (Wang et al., 2024b; Ni et al., 2026). Our focus is how to coordinate an already supplied skill set for a shared task.

Planning and skill preparation. Planning and compilation introduce structure into model calls and tool execution (Yao et al., 2022; Wang et al., 2024a; Kim et al., 2023; Khattab et al., 2024). For packaged skills, SkillRouter and Graph-of-Skills improve routing and dependency-aware retrieval (Zheng et al., 2026; Liu et al., 2026). AgentSkillOS couples discovery with graph-based orchestration and artifact handoffs (Li et al., 2026a); SkVM adapts skills to model capabilities and environments through compilation and runtime optimization (Chen et al., 2026). SkillRAE generates task-specific guidance in an auxiliary context while retaining the original packages (Meng et al., 2026). SkillCompiler jointly compiles the supplied skill set into a single task-specific package, coordinating guidance and resource choices before execution.

3 SkillsBench-Compose

SkillsBench-Compose evaluates whether agents can satisfy cross-component relationships while meeting each component's own requirements. Its central design is to retain component evaluation and add separate checks of the relationships needed to achieve a shared goal.

3.1 Task definition

Each instance specifies both the component tasks and the relationships that must hold between them. We represent it as

$$\mathcal{T} = (x, \mathcal{S}, \mathcal{C}, \mathcal{R}), \tag{1}$$

where x is the unified task specification, $\mathcal{S}$ is the set of supplied skill packages, $\mathcal{C} = \{c_1, \ldots, c_k\}$ contains the component tasks, and $\mathcal{R}$ specifies their required relationships. These relationships include artifact handoffs, shared objects or state, and joint constraints on outputs. The agent receives the task specification and skill packages in the task environment; its submission is evaluated against both component requirements and cross-component relationships. Each instance contains two or three component tasks, each of which may require several skills.

For example, in the dialogue catalog task, component checks evaluate parsing and retrieval separately, while integration checks additionally verify that search hits refer to the correct nodes in the parsed story graph (Appendix B.1).

3.2 Benchmark construction

The benchmark covers three forms of cross-component relationship, with artifact handoffs accounting for the largest share (Figure 1a). Its 20 tasks comprise 17 two-component and three three-component instances, each supplying 3–11 distinct source skill packages (Figure 1b).

We selected component tasks from SkillsBench whose inputs, environments, and objectives could support an explicit relationship, then wrote a unified specification and adapted the inputs and evaluation programs to the shared goal. We retained the original skill packages and component acceptance requirements, tolerances, and reward definitions, while parameterizing inputs, entities, and output paths as needed. Separate integration checks test the added relationships. Each human-authored task received final approval from three independent reviewers, and the task set was frozen before method evaluation.

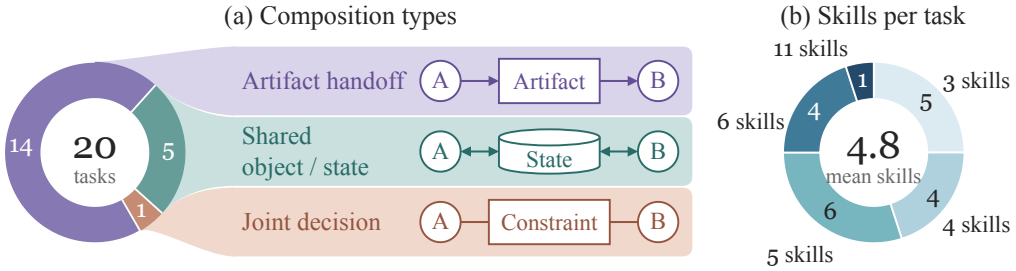


Figure 1: Task composition in SkillsBench-Compose. (a) Relationship types, using one frozen label per task; A and B denote task components. (b) Distinct source skill packages per task. Numbers on the rings count tasks; the right center reports the mean number of skills per task. Both distributions cover 20 tasks.

Validation tests the distinction between component success and complete solutions: all 20 reference solutions pass every check, and each task includes at least one constructed negative example that passes all component checks but fails integration. Appendix B details authoring, independent review, and the complete validation set.

3.3 Evaluation metrics

Full completion is the primary metric because a composed task requires both its component functions and their relationships to be correct. For a submission y, let $p_j(y) \in \{0, 1\}$ indicate whether component c_j satisfies its predefined acceptance conditions, and let $I(y) \in \{0, 1\}$ indicate whether all required integration checks pass. Define all-component acceptance A and full completion F as

$$A(y) = \prod_{j=1}^{k} p_j(y), \qquad F(y) = A(y)I(y). \tag{2}$$

Our primary metric is the full completion rate, the mean of F over evaluated submissions. We also report the all-component pass rate (mean A), integration pass rate (mean I), and mean component reward. For the latter, we average original component rewards within each submission and then average equally over submissions. Section 5 specifies the repeated-evaluation protocol. The integration pass rate measures relational correctness; full completion additionally requires every component to pass.

4 SkillCompiler: Task-Specific Joint Compilation

SkillCompiler makes cross-skill coordination explicit before execution by jointly specializing instructions and selecting resources for the task. The relationships in Section 3 couple these choices: a producer's output format constrains what a consumer can read, and the retained resources must support the resulting workflow.

Inspired by program compilation, SkillCompiler separates a model's semantic decisions from programmatic validation and packaging. The model produces a structured intermediate representation (IR) of task-specific guidance, rules, and resource choices. A program checks its structure and references and assembles the guidance with selected source resources into a skill package for the unchanged executor (Figure 2).

4.1 Intermediate representation

The IR links the generated workflow to the source rules and resources needed to execute it. Given the task specification x and relevant skill packages $\mathcal{S}$, SkillCompiler constructs $\mathrm{Index}(\mathcal{S})$ from each package's full `SKILL.md`, resource metadata, and bounded file previews. The index exposes resource locations and interfaces for selection; Appendix C specifies the input contents. For multiple

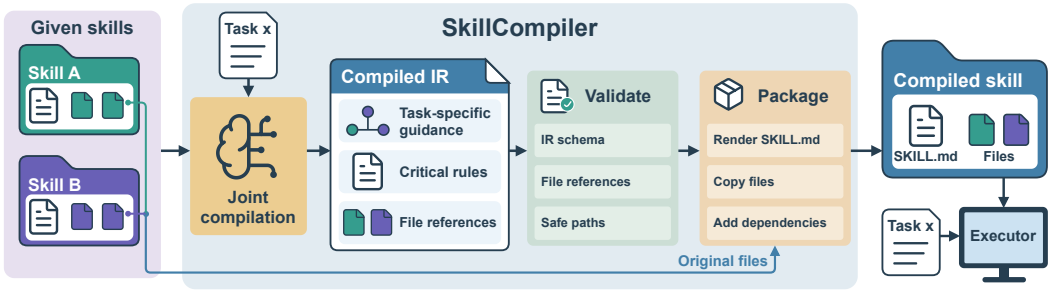


Figure 2: SkillCompiler prepares a task-specific package before execution. Joint compilation produces an IR containing coordinated guidance, critical source rules, and file references. Programmatic validation checks its structure and references; packaging renders `SKILL.md` and copies selected original files with their identified local dependencies. The executor receives the resulting package and the original task. Resource colors indicate their origins.

skills, the compiler model f_θ produces the IR through a structured call; after validation, a program constructs the package:

$$z = (g, \mathcal{K}, \mathcal{F}) = f_\theta(x, \mathrm{Index}(\mathcal{S})), \qquad s_x = \mathrm{Package}(z, \mathcal{S}). \tag{3}$$

Here g specifies a coordinated workflow, $\mathcal{K}$ records critical source rules paired with task-specific execution guidance, and $\mathcal{F}$ specifies selected files, their package paths, and their intended use. The normal path uses one semantic model call, with the same backbone as the executor. A single supplied skill is returned unchanged.

4.2 Joint Compilation

Joint compilation organizes the supplied procedures around shared outputs and dependencies. The compiler considers each skill's instructions in the context of the others and is prompted to identify the task's required outputs, constraints, branches, and completion criteria. The resulting workflow specifies which step produces each intermediate artifact, how later steps consume it, and which conditions govern the handoff.

Source rules constrain this specialization. For each critical rule, the IR records both the original requirement and its task-specific application, including relevant parameters, interfaces, and checks. The prompt distinguishes obligations from recommendations. Rendering the rule alongside its application keeps the general requirement visible with the concrete execution guidance.

Resource choices connect the workflow to the implementations it will use. When a required mechanism is already implemented in a source script, the compiler can retain that file and describe its interface, avoiding a separate natural-language reimplementation. References and templates can similarly be retained for the executor to inspect when needed. Their intended uses and destination paths are recorded in $\mathcal{F}$ and coordinated with the workflow. Before returning the IR, the model is prompted to check requirement coverage, fidelity to source rules, executable branches, and reference consistency (Appendix E).

4.3 Package Construction

Package construction combines the generated guidance and its referenced resources into a skill that the existing executor can load. Programmatic validation checks the IR's schema, source-file references, safe destination paths, and packaging feasibility. The packager directly renders the workflow, retained rules, and resource-use instructions into `SKILL.md`, copies selected original files to the paths specified in the IR, and includes their identified local dependencies. The executor receives the original task and the compiled package as its sole skill input, while retaining its existing tools and execution loop.

Structural and packaging-precheck errors trigger at most two repair generations conditioned on the reported errors. These calls occur before execution and are included in preparation cost. Appendix C

specifies the field mapping, recovery inputs, and Raw fallback protocol for unresolved preparation failures.

5 EXPERIMENTS

SkillCompiler achieves the highest overall SkillsBench reward and Compose full completion with all three backbones. Ablations compare task-only guidance, independent rewriting, and joint preparation to explain the value of the compilation design; the cost analysis accounts for both preparation and execution.

5.1 EXPERIMENTAL SETUP

Benchmarks and metrics. We evaluate on 87 SkillsBench tasks (Li et al., 2026b) and the 20 tasks in SkillsBench-Compose. SkillsBench uses the original task rewards, averaged equally across tasks and reported on a 0–100 scale. We report the overall mean and groups with one, two to three, and at least four supplied skills (23, 43, and 21 tasks). SkillsBench-Compose reports mean component reward, all-component pass rate, integration pass rate, and full completion rate (Section 3); full completion is its primary metric.

Models and execution. We use GPT-5.4 Mini, GPT-5.4, and Qwen3.7-plus with OpenHands. The two GPT models use the `xhigh` reasoning setting, and Qwen3.7-plus uses `high`. SkillCompiler uses the same backbone for preparation and execution. All skill-based methods receive the same task-associated skill set, so the comparison evaluates how they use the supplied skills.

Baselines. **No-skill** executes the task without skill packages; **Raw** provides the original packages; and **Concat** joins their instructions while retaining their resources. We also compare with **SkVM-AOT**, the ahead-of-time preparation variant of SkVM (Chen et al., 2026); **AgentSkillOS**, using its Quality-First Oracle configuration with OpenHands execution nodes (Li et al., 2026a); and **Skill-RAE**, which prepares an auxiliary context layer while retaining the original packages (Meng et al., 2026). These methods are adapted to the supplied skill set. Appendix D describes the adaptations and execution limits.

Evaluation protocol. SkillsBench uses one valid execution per task and condition. SkillsBench-Compose uses three executions per task, with independent preparations across the three runs. We report means over all 60 executions. On single-skill tasks, Concat and SkillCompiler are identity transformations and reuse Raw. Unresolved preparation failures use the matched Raw execution, with failed preparation included in resource accounting. All methods and backbones use the same task-specific execution limit of 4 or 8 hours on Compose.

5.2 MAIN RESULTS

Joint compilation improves overall performance across backbones. SkillCompiler achieves the highest primary score for all three backbones on both benchmarks (Table 2). Relative to the strongest baseline for each backbone, it improves mean reward by 2.94–4.83 points on SkillsBench and full completion by 3.33–8.33 percentage points on SkillsBench-Compose. The gains thus extend from average task reward to complete solutions satisfying both component and integration checks.

The SkillsBench gains come from multi-skill tasks. SkillCompiler reuses Raw for single-skill tasks and obtains the highest mean reward in both the 2–3-skill and four-or-more-skill groups for every backbone. Its advantage over Concat extends to the four-or-more-skill group, where Raw and Concat yield their smallest gains over No-skill (Table 1).

Compose exposes a gap between component success and full completion. SkillCompiler leads all four Compose metrics for each backbone, improving component performance as well as integration. With GPT-5.4 and Concat, six of 60 executions pass every component but fail integration. Compose therefore detects relationship errors hidden by component-level success. The highest full completion rates are 20.00–26.67%, making Compose a challenging test of coordinated skill use.

Table 2: Main results (higher is better). SkillsBench reports mean reward over 87 tasks (23/43/21 by skill count). Compose reports mean component reward (Comp.), all-component pass rate (All-comp.), integration pass rate (Integr.), and full completion rate (Full) over 20 tasks with three independent preparation-and-execution runs each. Rewards use a 0–100 scale; pass rates are percentages. Bold and underline mark the best and second-best values within each backbone and column, including ties. Single-skill Concat and SkillCompiler reuse Raw. Shading highlights SkillCompiler and the primary metrics, Overall and Full.

	SkillsBench				**SkillsBench-Compose**			
Method	1 skill	2–3	4+	**Overall**	Comp.	All-comp.	Integr.	**Full**
GPT-5.4 Mini								
No-skill	42.03	27.44	30.48	32.03	28.15	6.67	30.00	6.67
Raw	<u>56.52</u>	40.58	37.62	44.08	32.83	11.67	<u>36.67</u>	10.00
Concat	<u>56.52</u>	40.58	23.81	40.75	<u>46.50</u>	<u>28.33</u>	<u>36.67</u>	<u>18.33</u>
SkVM-AOT	**65.22**	43.60	<u>42.86</u>	49.14	35.15	25.00	33.33	13.33
AgentSkillOS	55.80	<u>57.37</u>	34.13	51.34	41.86	23.33	35.00	11.67
SkillRAE	55.65	55.81	38.10	<u>51.49</u>	39.79	<u>28.33</u>	33.33	15.00
SkillCompiler	<u>56.52</u>	**58.14**	**52.38**	**56.32**	**53.05**	**33.33**	**40.00**	**23.33**
GPT-5.4								
No-skill	33.33	39.07	49.84	40.15	31.54	13.33	25.00	10.00
Raw	56.52	55.63	43.65	52.97	42.91	16.67	28.33	15.00
Concat	56.52	55.12	50.16	54.29	35.83	21.67	28.33	11.67
SkVM-AOT	**65.22**	<u>56.63</u>	38.89	54.62	45.42	<u>23.33</u>	<u>38.33</u>	<u>18.33</u>
AgentSkillOS	**65.22**	45.53	47.62	51.24	<u>48.65</u>	<u>23.33</u>	33.33	13.33
SkillRAE	<u>63.04</u>	54.19	<u>52.86</u>	<u>56.21</u>	38.31	16.67	35.00	11.67
SkillCompiler	56.52	**62.23**	**55.71**	**59.15**	**55.96**	**36.67**	**45.00**	**26.67**
Qwen3.7-plus								
No-skill	27.08	30.93	28.57	29.34	24.82	8.33	23.33	3.33
Raw	49.82	50.12	19.05	42.54	39.82	15.00	<u>35.00</u>	13.33
Concat	49.82	<u>53.05</u>	29.05	46.40	26.90	10.00	26.67	8.33
SkVM-AOT	51.36	48.95	28.57	44.67	<u>41.07</u>	<u>21.67</u>	31.67	<u>16.67</u>
AgentSkillOS	**60.87**	50.12	23.81	46.61	34.82	15.00	26.67	11.67
SkillRAE	<u>60.14</u>	51.86	<u>38.10</u>	<u>50.73</u>	31.73	16.67	21.67	13.33
SkillCompiler	49.82	**59.19**	**48.57**	**54.15**	**51.87**	**26.67**	**40.00**	**20.00**

Appendix D.5 examines a concrete improvement and a remaining failure through the compiled guidance and execution evidence.

5.3 Ablation study

Joint compilation retains an advantage when independently rewritten skills are consolidated into one package. We evaluate all three backbones on the 64 multi-skill SkillsBench tasks and all 20 Compose tasks. **Task-only** generates guidance without source skills. **Per-Skill Rewrite** independently adapts each package to the full task, retaining separate packages. **Per-Skill Rewrite + Concat** merges the same generated guidance and resources into one package without another model call. Raw provides the original packages as a reference.

SkillCompiler outperforms Task-only with every backbone on both benchmarks (Table 3), supporting task-specific preparation that uses the supplied skill packages.

Consolidating the same independently rewritten artifacts also improves the aggregate primary score for every backbone on both benchmarks. Since the two per-skill variants reuse the same generated instructions and resources, their comparison tests the organization of that content at execution time. The gains favor making the adapted skills available together in a common package.

Table 3: Ablations with Raw as a reference. SkillsBench reports mean reward on 64 multi-skill tasks (43 with 2–3 skills; 21 with 4+); Multi averages all 64. Compose reports component reward (Comp.), all-component pass (All-comp.), integration pass (Integr.), and full completion (Full) over 20 tasks with three runs each. Rewards use a 0–100 scale; pass rates are percentages. Bold and underline mark the best and second-best values within each backbone and column, including ties. The two per-skill variants share preparation matched by task and run, with separate executions.

	SkillsBench			**SkillsBench-Compose**			
Method	2–3	4+	**Multi**	Comp.	All-comp.	Integr.	**Full**
GPT-5.4 Mini							
Raw	40.58	37.62	39.61	32.83	11.67	<u>36.67</u>	10.00
Task-only	29.49	30.95	29.97	30.73	<u>18.33</u>	28.33	10.00
Per-Skill Rewrite	45.23	36.98	42.53	35.80	13.33	25.00	6.67
Per-Skill Rewrite + Concat	<u>47.91</u>	<u>41.49</u>	<u>45.80</u>	<u>38.76</u>	<u>18.33</u>	28.33	<u>11.67</u>
SkillCompiler	**58.14**	**52.38**	**56.25**	**53.05**	**33.33**	**40.00**	**23.33**
GPT-5.4							
Raw	55.63	43.65	51.70	42.91	16.67	28.33	<u>15.00</u>
Task-only	42.18	44.70	43.01	38.13	18.33	28.33	11.67
Per-Skill Rewrite	<u>56.61</u>	51.70	55.00	43.11	20.00	30.00	11.67
Per-Skill Rewrite + Concat	56.60	<u>52.84</u>	<u>55.36</u>	<u>47.68</u>	<u>21.67</u>	<u>33.33</u>	<u>15.00</u>
SkillCompiler	**62.23**	**55.71**	**60.09**	**55.96**	**36.67**	**45.00**	**26.67**
Qwen3.7-plus							
Raw	50.12	19.05	39.92	39.82	15.00	<u>35.00</u>	<u>13.33</u>
Task-only	48.70	29.65	42.45	41.31	13.33	28.33	10.00
Per-Skill Rewrite	53.88	34.30	47.45	39.03	15.00	30.00	10.00
Per-Skill Rewrite + Concat	<u>54.95</u>	<u>37.85</u>	<u>49.34</u>	<u>43.49</u>	<u>16.67</u>	<u>35.00</u>	<u>13.33</u>
SkillCompiler	**59.19**	**48.57**	**55.70**	**51.87**	**26.67**	**40.00**	**20.00**

Joint compilation further improves on this consolidated variant by 4.73–10.45 reward points on SkillsBench and 6.67–11.67 percentage points in Compose full completion. These gains support preparing skills jointly beyond placing independently generated instructions in a single package. Its advantage extends to both skill-count groups and all four Compose metrics, with integration pass rates improving by 5.00–11.67 percentage points across the three backbones. Higher component rewards and all-component pass rates accompany these integration gains. Thus, joint preparation improves both requirements for full completion: carrying out component procedures and satisfying the relationships between them. Its advantage in the 2–3-skill group also shows that joint preparation is useful with a small supplied skill set, where a shared task can already require coordination between distinct procedures. Resource usage and instruction lengths appear in Appendices D.3 and D.4.

5.4 Resource cost

SkillCompiler improves the primary score while using fewer total tokens than the highest-scoring baseline in each of the six model–benchmark settings (Figure 3). On SkillsBench, total usage is 16.3–37.9% lower than SkillRAE. On Compose, it is 5.4% lower than Concat for Mini and 31.3% and 13.1% lower than SkVM-AOT for GPT-5.4 and Qwen3.7-plus. These comparisons include preparation as well as execution. We sum model input and output tokens across the calls associated with each execution. For SkillCompiler, every Compose run includes a fresh compilation. The comparison therefore accounts for constructing the package as well as the subsequent interactions through which the agent uses it.

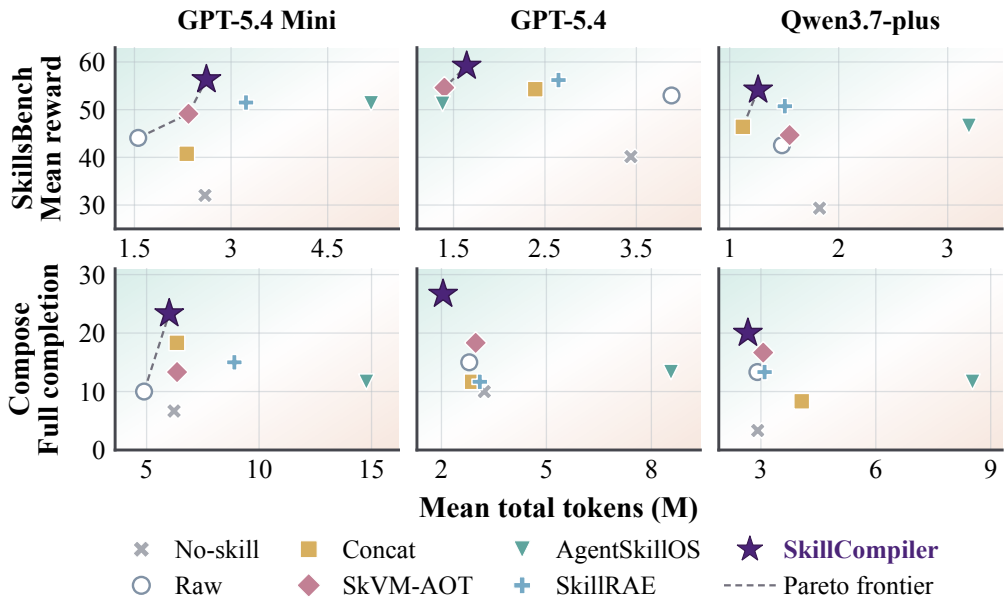


Figure 3: Primary score versus mean total model tokens per execution, including preparation (M denotes millions). Compose full completion is shown as a percentage. Dashed segments connect observed Pareto-optimal configurations within each panel; shading indicates the preferred direction (higher and further left). Horizontal scales differ across panels. SkillsBench averages 87 task executions; Compose averages 60 executions over 20 tasks.

Preparation is a small part of SkillCompiler's total model usage, accounting for 1.25–3.28% across these settings (Appendix D.2). Preparation succeeds without Raw fallback in all evaluated SkillCompiler runs, with 1.00–1.13 generation calls per preparation, including bounded repair.

Joint compilation also reduces preparation usage relative to independent rewriting. Compared with Per-Skill Rewrite + Concat on the 64 multi-skill SkillsBench tasks and all Compose executions, it uses 77.7–87.5% fewer preparation tokens across the six settings. Independent rewriting repeats the full task context for each source package. Joint compilation instead processes the skills together, sharing that context while generating a single workflow and selecting its resources. Total usage is lower in five of these comparisons; Mini on SkillsBench is the exception, using 11.7% more total tokens (Appendix D.3).

Relative to Raw, SkillCompiler's higher scores come with lower total token usage for GPT-5.4 and Qwen3.7-plus and higher usage for Mini on both benchmarks. Appendix D.2 reports stage-level usage for all methods and preparation-call counts.

6 Conclusion

We studied how agents jointly use relevant skills once they have been supplied. SkillsBench-Compose evaluates component completion and integration correctness separately, exposing composition failures that component checks alone miss. SkillCompiler jointly prepares guidance and source resources into a task-specific package before execution, using the existing executor. Across three backbones on SkillsBench and SkillsBench-Compose, it achieves higher primary scores while using fewer total tokens, including preparation, than the highest-scoring baseline in each setting. Ablations show an advantage for joint compilation even when independently rewritten skills are consolidated into a single package. These results support task-specific joint preparation as an effective way to organize reusable skills into workflows that serve a shared goal. Broader task coverage and adaptation during execution remain directions for future work (Appendix A).

AI USE STATEMENT

Generative AI tools were used to aid and polish manuscript writing, including improvements to clarity, wording, grammar, and presentation. LLMs were also used as components of the research and experimental pipeline, as described in the paper. During the construction of SkillsBench-Compose, LLMs assisted candidate discovery, prototype implementation, and preliminary technical checks. Human authors modified and verified the final tasks, reference solutions, and evaluation programs, and each retained task was independently reviewed by three human reviewers without LLM assistance. The authors reviewed all AI-assisted content and experimental outputs and take responsibility for the final text, analyses, and artifacts.

REPRODUCIBILITY STATEMENT

§3 and Appendix B describe SkillsBench-Compose construction, independent review, validation, and evaluation metrics, including a complete example task. §4 and Appendix C specify SkillCompiler's representation, compilation, validation, and packaging; Appendix E provides the compiler and ablation prompts, message templates, and response schema. §5 and Appendix D document model settings, baseline adaptations, execution limits, repeated evaluation, and resource accounting. Appendix D.5 includes compiled skill excerpts and corresponding execution and verification evidence.

REFERENCES

Léo Boisvert, Megh Thakkar, Maxime Gasse, Massimo Caccia, Thibault L De Chezelles, Quentin Cappart, Nicolas Chapados, Alexandre Lacoste, and Alexandre Drouin. Workarena++: Towards compositional planning and reasoning-based common knowledge work tasks. *Advances in Neural Information Processing Systems*, 37:5996–6051, 2024.

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large language models as tool makers. In *International Conference on Learning Representations*, volume 2024, pp. 54067–54089, 2024.

Le Chen, Erhu Feng, Yubin Xia, and Haibo Chen. Skvm: Revisiting language vm for skills across heterogenous llms and harnesses. *arXiv preprint arXiv:2604.03088*, 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.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Saiful Haq, Ashutosh Sharma, Thomas Joshi, Hanna Moazam, Heather Miller, et al. Dspy: Compiling declarative language model calls into state-of-the-art pipelines. In *International Conference on Learning Representations*, volume 2024, pp. 54928–54958, 2024.

Sehoon Kim, Suhong Moon, Ryan Tabrizi, Nicholas Lee, Michael W Mahoney, Kurt Keutzer, and Amir Gholami. An llm compiler for parallel function calling. *arXiv preprint arXiv:2312.04511*, 2023.

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, Yimin Liu, Wenbo Chen, Bingran You, Zonglin Di, Yifeng He, Shenghan Zheng, Kyoung Whan Choe, Jiankai Sun, Shuyi Wang, et al. Skillsbench: Benchmarking how well agent skills work across diverse tasks. *arXiv preprint arXiv:2602.12670*, 2026b.

Dawei Liu, Zongxia Li, Hongyang Du, Xiyang Wu, Shihang Gui, Yongbei Kuang, and Lichao Sun. Graph-of-skills: Dependency-aware structural retrieval for massive agent skills. *arXiv preprint arXiv:2604.05333*, 2026.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. Agentbench: Evaluating llms as agents. In *International Conference on Learning Representations*, volume 2024, pp. 52989–53046, 2024.

Xiangcheng Meng, Shu Wang, and Yixiang Fang. Skillrae: Agent skill-based context compilation for retrieval-augmented execution. *arXiv preprint arXiv:2605.10114*, 2026.

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. *arXiv preprint arXiv:2603.25158*, 2026.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. Toolllm: Facilitating large language models to master 16000+ real-world apis. In *International Conference on Learning Representations*, volume 2024, pp. 9695–9717, 2024.

Yongliang Shen, Kaitao Song, Xu Tan, Wenqi Zhang, Kan Ren, Siyu Yuan, Weiming Lu, Dongsheng Li, and Yueting Zhuang. Taskbench: Benchmarking large language models for task automation. *Advances in Neural Information Processing Systems*, 37:4540–4574, 2024.

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.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. Appworld: A controllable world of apps and people for benchmarking interactive coding agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 16022–16076, 2024.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents. *arXiv preprint arXiv:2402.01030*, 2024a.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. *arXiv preprint arXiv:2409.07429*, 2024b.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, et al. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems*, 37:52040–52094, 2024.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*, 2024.

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, JiHuai Zhu, Yong Wu, Tianze Xu, Baohua Dong, Hangcheng Zhu, Ruohui Huang, et al. Skillrouter: Skill routing for llm agents at scale. *arXiv preprint arXiv:2603.22455*, 2026.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. Webarena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations*, volume 2024, pp. 15585–15606, 2024.

A LIMITATION

Benchmark coverage. SkillsBench-Compose focuses on tasks with explicit component goals and cross-component relationships, allowing component completion and integration correctness to be evaluated separately. The current instances contain two or three components and cover artifact handoffs, shared objects or state, and joint constraints. Dependencies spanning larger sets of components remain to be evaluated. Future extensions can broaden the task domains and composition structures while retaining separate component and integration checks.

Scope of compilation. SkillCompiler focuses on preparing supplied relevant skills before execution. Programmatic checks validate the structure and resource references of the compiled package; preserving the meaning of source instructions and carrying out the coordinated workflow still depend on the compiler model and executor. The current compiler does not revise the package using execution feedback, leaving the executor to respond to new observations. Combining joint compilation with skill retrieval and updates informed by execution feedback is a direction for handling changes in task requirements or the available skill set.

B SKILLSBENCH-COMPOSE CONSTRUCTION DETAILS

Authoring and independent review. LLMs assisted candidate discovery, prototype implementation, and preliminary technical checks during construction. Human authors modified and checked the final tasks and were responsible for the resulting specifications, reference solutions, and evaluation programs. These model-assisted checks were separate from the final review. Each task was independently reviewed by three people who had not authored or modified the reviewed version. Reviewers read the materials and made their judgments without LLM assistance. All retained tasks received final approval from all three reviewers.

Packages and task adaptation. Repeated skill packages are deduplicated without changing their contents. When packages share a name but differ in version, compatibility review selects one complete source version. Task-specific adaptations include parameterizing inputs, entities, and output paths, as well as environment isolation, service lifecycle handling, and necessary reference-program corrections. These adaptations preserve the component acceptance requirements, tolerances, and reward definitions.

Validation cases. The validation set contains 95 scenarios: 43 pass all checks, including the 20 reference solutions and valid equivalent transformations; 46 pass all component checks but fail integration; and six cover other boundary or initial-state conditions in which not all components pass. Every task has at least one negative example of the second kind. These cases test acceptance of valid solutions and rejection of relationship errors that component checks alone do not detect. All are constructed validation cases, separate from agent evaluation outcomes.

Component acceptance. Full completion uses each component's predefined acceptance conditions. The Civ6 component passes at a reward of at least 0.95 together with all formatting and legality checks. This threshold was fixed before evaluation from the original scoring rule's upper bound of 19 and normalization denominator of 20; its reward is not rescaled to one. Other components retain their original full-pass conditions.

B.1 EXAMPLE TASK: SEARCHABLE DIALOGUE CATALOG

The searchable dialogue catalog combines dialogue parsing with parallel TF-IDF search. It supplies four skills: `dialogue-graph`, `memory-optimization`, `python-parallelization`, and `workload-balancing`. The components retain their respective acceptance conditions, while the catalog must connect search results to the submitted dialogue graph.

Integration requirements. The catalog reads the submitted graph, constructs documents using a specified graph-to-document mapping, and calls the submitted parallel indexer and batch searcher. Integration requires agreement among the retrieved document IDs, graph node IDs, original dialogue

text, and paths through that same graph. The full specification below makes these interfaces and relationships explicit.

Validation. The reference solution passed both component evaluations and all integration checks. A constructed `broken-link` variant also received a reward of 1.0 on each component, but passed only three of four integration tests; it therefore failed integration and full completion. These construction-time validation cases illustrate a distinction that component rewards alone miss.

Task specification. The task specification is reproduced in full below, with layout adapted for readability.

> **Searchable Dialogue Catalog**
>
> Deliver a searchable catalog for the branching dialogue in `/app/script.txt`. A search hit must identify the original dialogue node, show its unmodified dialogue text, and provide a valid route through the story to that node. The catalog must use a parallel TF-IDF engine that remains equivalent to the supplied sequential engine and meets its multicore performance requirements.
>
> **Dialogue Graph**
>
> Implement `parse_script(text: str)` in `/app/solution.py`. It must accept script content, not a filename, and return a graph dictionary or a graph object exposing that dictionary through `to_dict()`. Write `/app/dialogue.json` and `/app/dialogue.dot` from the provided script.
>
> The JSON graph has `nodes` and `edges` lists. Each node has string fields `id`, `text`, `speaker`, and `type`, where `type` is `line` or `choice`. Each edge has string fields `from`, `to`, and `text`. Preserve the script's dialogue, choices, and transitions. All nodes must be reachable from the first node, `Start`. Edge targets must exist, with the permitted terminal target `End`. `End` is a terminal sentinel, not an additional entry in `nodes`. Multiple paths can lead to `End`. The DOT file must visualize the same graph, including the distinction between dialogue and choice nodes.
>
> **Parallel Search Engine**
>
> The original sequential TF-IDF implementation and document generator are in `/root/workspace/`. Do not modify these reference files. Implement the following interfaces in `/root/workspace/parallel_solution.py`:
>
> - `build_tfidf_index_parallel(documents, num_workers=None, chunk_size=500)`, returning a `ParallelIndexingResult` with the original `TFIDFIndex` structure.
> - `batch_search_parallel(queries, index, top_k=10, num_workers=None, documents=None)`, returning `(List[List[SearchResult]], elapsed_time)`.
>
> Index values, ranked document IDs, and scores must agree with the original sequential implementation. Preserve its tokenization, weighting, normalization, and tie behavior. With four workers, achieve at least 1.5x sequential indexing speed and 2x sequential batch-search speed on the supplied generated workload. These performance requirements apply to the original benchmark-scale workload, not to the much smaller dialogue graph.
>
> **Catalog Integration**
>
> Implement the importable function `search_dialogue(graph_path, queries, top_k=5)` in `/app/catalog.py`. It must read the graph at `graph_path` at call time, build its corpus through the submitted parallel indexing interface, and answer the supplied list of query strings through the submitted parallel batch-search interface. Use four workers for both calls. The function must not depend on precomputed query answers or a particular filename.
>
> Use this deterministic graph-to-document mapping:
>
> 1. Sort the graph's nodes by their string IDs. Their zero-based positions are the integer `doc_id` values.
> 2. The document title is the node's `speaker`. Its content is the newline-joined sequence of the node's `text` followed by the `text` of every outgoing edge, preserving edge order in the graph. Do not add node IDs or target IDs to the indexed title or content.

Table 4: IR field mapping. The fields are generated jointly and are materialized into one executor-visible package.

Field	Content and use
`skill_md`	Coordinated guidance g: scope, ordered steps, conditions, artifact handoffs, and completion checks.
`critical_contracts`	Rules $\mathcal{K}$: each source rule is paired with task-specific execution guidance and internal provenance. The rule and guidance are rendered into `SKILL.md`.
`selected_files`	Resources $\mathcal{F}$: source references, destination paths, intended use, and the cost of omitting a file. Selected files and identified local dependencies are copied; their paths and usage are listed in `SKILL.md`.

3. Use the supplied `Document` type. Set `topic` to `dialogue`; set `word_count` to the number of whitespace-separated words in the content.

Return one object per query in input order, with `query` and `hits` fields. Each hit must have `doc_id`, `score`, `node_id`, `text`, and `path`:

- Preserve the TF-IDF result order and scores, with score agreement within `1e-6` of the sequential implementation.
- `node_id` must identify the same node as `doc_id` in the indexed graph.
- `text` must equal that node's original `text`, without adding choice labels or summarizing its contents.
- `path` is a list of node IDs beginning with `Start` and ending with the hit node. Every consecutive pair must be an edge in that same graph. Any valid path is accepted; a shortest path is not required.

Queries with no matches return an empty `hits` list. The catalog must work for arbitrary supplied query strings and top-k values, including an empty query string and repeated queries, using the same search semantics as the original engine. Keep all original dialogue and search-engine deliverables in addition to the integrated catalog.

C SkillCompiler Implementation Details

Full prompts and the response schema are provided in Appendix E.

Compiler inputs. The compiler receives the complete task specification and the full `SKILL.md` of each supplied package. Resource metadata includes paths, types, sizes, bounded text previews, available entry-point or docstring information, and statically identified local dependencies. Large scripts and reference documents are not loaded in full by default; binary resources are represented by metadata. A needed resource can be selected for later use; the prompt prohibits inventing instructions from unseen file contents. The compiler does not explore the target environment or access hidden evaluation programs, task trajectories, or rewards.

Specialization requirements. Task analysis, source-rule specialization, instruction generation, cross-skill coordination, and the output check are requirements within the same semantic call. The prompt requires preserving the conditions and force of source rules: an illustrative value must not become a mandatory threshold, and an optimization objective must not be reduced to merely meeting a threshold. These are generation requirements; the structural checks do not establish that the generated guidance satisfies them.

Execution interface. For multi-skill tasks, the executor sees the original task, its workspace, and the compiled package. The package contains a single `SKILL.md` and the selected runtime files, including supporting resources, as specified by the IR fields in Table 4. Internal source mappings, compiler inputs and outputs, validation records, and attempt logs are kept separately and are not exposed to the executor. For a single skill, SkillCompiler performs the identity transformation and reuses the matched Raw execution under the same model and harness; it does not invoke the compiler or run an additional executor episode.

Bounded recovery and failure accounting. There is one initial semantic call and at most two additional generations for invalid JSON, schema violations, invalid file references or paths, or failed packaging prechecks. A recovery call receives the original compiler input, the preceding output, and the structural errors. It receives no task-execution feedback. If actual filesystem operations fail after a valid IR has been obtained, that IR is retained and the failure is reported; the filesystem failure does not trigger another semantic call.

If preparation still fails, the main evaluation uses the matched Raw result from the same model, harness, and trial, without rerunning Raw or conditioning fallback on the executor's score. Failed preparation calls and the reused Raw execution are included in end-to-end usage. Preparation coverage and a strict score assigning zero to unresolved preparation failures are retained alongside this fallback score. Single-skill identity is separate from failure fallback.

D Experimental Details

D.1 Baseline adaptation and protocol

Fixed supplied skills. All skill-based methods start with the packages associated with the task. Retrieval from an external skill library and rescue by adding other skills are excluded. Raw leaves the packages unchanged. Concat joins their instructions and preserves the resources; single-skill instances reuse Raw. SkillCompiler likewise reuses Raw for single-skill instances and compiles only multi-skill inputs.

External methods. We evaluate SkVM's ahead-of-time preparation mode, denoted SkVM-AOT, rather than its runtime preparation variant. AgentSkillOS uses its Quality-First Oracle configuration, with native dependency-graph planning and OpenHands execution at each node. Its planning and node-level calls count as execution usage. Our SkillRAE adaptation retains the original packages and adds a `READ_FIRST.md` context layer before the task. It uses structural Markdown extraction for the extraction rules not released with the method, BGE ranking over 30 candidate units, at most three cues per package, and a 384-token coordination budget. The input set remains fixed; no cross-library retrieval is used. This offline extraction makes no generative-model preparation calls.

Execution limits. SkillsBench timeout extensions use at least 7,200 seconds, retaining a task's original limit when longer. Only previously truncated executions were rerun; valid executions that finished under the earlier 3,600-second setting were retained. On SkillsBench, AgentSkillOS uses a 3,600-second node timeout, a 32,400-second graph timeout, and up to six concurrent nodes per task; Mini uses a 900-second request timeout. All formal Compose evaluations, including the ablations and AgentSkillOS, use task-specific execution limits of 14,400 or 28,800 seconds. For a given task, the limit is the same across methods, backbones, and all three runs. Compiler repair is bounded and occurs before execution (Appendix C); it does not use execution outcomes to revise a skill.

Repeated evaluation and failures. SkillsBench has one valid execution for each task, method, and model. Compose has three runs per task and condition, with independent preparation across runs. Per-Skill Rewrite + Concat reuses the corresponding Per-Skill Rewrite artifacts for each task and run, then executes the merged package separately; no preparation is shared across runs. The primary table averages all 60 resulting executions, including valid failures and time-limit truncations. Infrastructure interruptions are recovered to obtain a valid execution rather than scored as zero; a valid low score does not trigger resampling. If bounded preparation fails, evaluation uses the matched Raw execution for that task, model, and run. Preparation usage is retained when reporting the total.

D.2 Resource accounting

We sum model input and output tokens across calls, then average over the same task executions as the corresponding score (Table 5). Total usage includes both preparation and execution. Shared preparation is amortized over the evaluated task set and included in preparation, with the same allocation retained in skill-count subsets. Failed preparation and bounded recovery calls are included. Cached input is a subset of input tokens and is not added a second time. The reported tokens measure

Table 5: Mean model token usage per execution, in thousands, for all main methods. SkillsBench uses 87 executions and Compose uses 60 executions of 20 tasks. Shared preparation is included in Prep.; AgentSkillOS planning is included in Exec. Independently rounded columns can differ slightly from their displayed sum.

	SkillsBench			SkillsBench-Compose		
Method	Prep.	Exec.	Total	Prep.	Exec.	Total
GPT-5.4 Mini						
No-skill	0.0	2595.2	2595.2	0.0	6218.6	6218.6
Raw	0.0	1558.3	1558.3	0.0	4887.2	4887.2
Concat	0.0	2312.1	2312.1	0.0	6347.9	6347.9
SkVM-AOT	297.5	2041.7	2339.2	738.4	5615.7	6354.1
AgentSkillOS	0.0	5174.7	5174.7	0.0	14770.9	14770.9
SkillRAE	0.0	3231.8	3231.8	0.0	8900.8	8900.8
SkillCompiler	33.3	2583.9	2617.2	112.0	5892.8	6004.7
GPT-5.4						
No-skill	0.0	3440.0	3440.0	0.0	3224.9	3224.9
Raw	0.0	3884.5	3884.5	0.0	2795.0	2795.0
Concat	0.0	2393.8	2393.8	0.0	2877.0	2877.0
SkVM-AOT	234.9	1165.2	1400.1	385.8	2592.2	2978.0
AgentSkillOS	0.0	1380.3	1380.3	0.0	8560.9	8560.9
SkillRAE	0.0	2648.2	2648.2	0.0	3096.8	3096.8
SkillCompiler	26.8	1617.0	1643.8	67.2	1979.7	2047.0
Qwen3.7-plus						
No-skill	0.0	1824.7	1824.7	0.0	2916.4	2916.4
Raw	0.0	1479.6	1479.6	0.0	2908.6	2908.6
Concat	0.0	1120.8	1120.8	0.0	4066.0	4066.0
SkVM-AOT	135.0	1415.1	1550.1	397.0	2663.0	3060.0
AgentSkillOS	0.0	3194.9	3194.9	0.0	8524.8	8524.8
SkillRAE	0.0	1506.0	1506.0	0.0	3096.4	3096.4
SkillCompiler	17.3	1243.6	1260.9	33.2	2627.0	2660.2

cumulative model usage, not wall-clock latency, monetary cost, or the context length of a single request. SkillRAE's non-generative offline extraction is not assigned generative-model tokens. Table 6 reports preparation success and mean generation calls, including bounded repair.

D.3 Ablation details

Task-only uses a separate prompt adapted to generating guidance without source skills. It shares the model, output schema, validation, and packaging pipeline with SkillCompiler. Its selected-file set is empty and its rule references are restricted to the task specification. The comparison thus tests access to source packages as a whole, including their procedures and resources. Per-Skill Rewrite + Concat mechanically merges the independently generated instructions and files, handling the paths needed to make the combined package usable. It makes no additional generative call and retains the matched per-skill preparation cost in its total usage.

Table 3 reports both skill-count groups and the multi-skill aggregate on SkillsBench, alongside all four Compose metrics. SkillCompiler leads every reported column within each backbone.

Relative to Per-Skill Rewrite + Concat, joint compilation uses 77.7–87.5% fewer preparation tokens across the six settings (Table 7). Total usage is also lower in five settings; with Mini on SkillsBench, it is 11.7% higher. Preparation savings therefore coexist with execution costs that depend on the task and backbone.

Table 6: Preparation outcomes and mean generative calls, including bounded repair. Success denotes a usable preparation output, not successful task execution. SkillsBench excludes the 23 single-skill identity cases for SkillCompiler and its ablations; SkVM-AOT prepares all 87 tasks. Every Compose task is prepared independently three times. Per-Skill Rewrite + Concat reuses the matched Per-Skill Rewrite preparation.

	SkillsBench			SkillsBench-Compose		
Method	Success	Fallback	Calls/run	Success	Fallback	Calls/run
GPT-5.4 Mini						
SkVM-AOT	52/87	35	7.68	34/60	26	13.78
Task-only	64/64	0	1.03	60/60	0	1.02
Per-Skill Rewrite	64/64	0	3.30	60/60	0	5.22
SkillCompiler	64/64	0	1.05	60/60	0	1.00
GPT-5.4						
SkVM-AOT	62/87	25	7.66	43/60	17	13.85
Task-only	64/64	0	1.06	60/60	0	1.05
Per-Skill Rewrite	64/64	0	3.45	60/60	0	5.25
SkillCompiler	64/64	0	1.13	60/60	0	1.08
Qwen3.7-plus						
SkVM-AOT	62/87	25	7.74	41/60	19	13.92
Task-only	64/64	0	1.00	60/60	0	1.02
Per-Skill Rewrite	64/64	0	3.41	60/60	0	5.13
SkillCompiler	64/64	0	1.06	60/60	0	1.00

Table 7: Mean token usage for the ablations, in thousands per execution. SkillsBench includes only its 64 multi-skill tasks; Compose includes 60 executions. Both per-skill rewriting variants include the same preparation cost.

	SkillsBench			SkillsBench-Compose		
Method	Prep.	Exec.	Total	Prep.	Exec.	Total
GPT-5.4 Mini						
Task-only	56.9	2916.1	2973.0	70.4	7759.8	7830.3
Per-Skill Rewrite	261.5	2329.1	2590.5	503.1	9448.1	9951.3
Per-Skill Rewrite + Concat	261.5	2574.1	2835.6	503.1	8167.3	8670.4
SkillCompiler	45.2	3122.2	3167.4	112.0	5892.8	6004.7
GPT-5.4						
Task-only	46.3	3894.6	3940.8	42.7	4157.6	4200.3
Per-Skill Rewrite	215.6	2193.7	2409.3	359.3	3778.8	4138.1
Per-Skill Rewrite + Concat	215.6	2453.5	2669.1	359.3	3420.3	3779.6
SkillCompiler	36.5	1441.5	1478.0	67.2	1979.7	2047.0
Qwen3.7-plus						
Task-only	25.3	1730.1	1755.4	21.7	3786.9	3808.6
Per-Skill Rewrite	159.7	1470.4	1630.1	266.5	5114.5	5381.0
Per-Skill Rewrite + Concat	159.7	1979.4	2139.1	266.5	5298.5	5565.0
SkillCompiler	23.5	1427.2	1450.7	33.2	2627.0	2660.2

Table 8: Mean available instruction length in tokens for GPT-5.4 Mini. SkillsBench covers 64 multi-skill tasks, split into 43 with 2–3 skills and 21 with 4+ skills. Compose uses the 20 artifacts from the original one-run-per-task evaluation.

	SkillsBench			Compose
Method	2–3 skills	4+ skills	All multi-skill	All tasks
Raw	4096	6581	4911	7026
Per-Skill Rewrite	3068	6133	4073	8785
SkillCompiler	1913	2031	1952	2068

D.4 GUIDANCE LENGTH

Joint compilation produces shorter static instructions than Raw or Per-Skill Rewrite in the measured artifacts (Table 8). We tokenize the full available `SKILL.md` text with `cl100k_base`, summing over packages for Raw and Per-Skill Rewrite. The count includes metadata, inline code examples, and resource paths, but excludes the task, common harness prompts, separate scripts, templates, and reference documents. It describes available guidance rather than the text actually loaded in any particular model call.

Per-Skill Rewrite + Concat holds the independently rewritten content fixed while changing its packaging. Joint compilation changes both the generated content and instruction length, so the comparison evaluates the preparation design as a whole.

D.5 EXECUTION CASES

The following cases connect generated skill content to recorded execution and verification. They come from the original one-run-per-task Compose evaluation under its recorded settings. Excerpts retain the original wording, with omissions marked and line breaks adjusted for readability.

Connecting CAD repair to review exports. On `accessible-cad-review-export`, GPT-5.4 with Concat passes both component checks but fails integration; SkillCompiler passes both components and integration. The task requires a repaired restroom CAD design and Three.js/OBJ exports depicting that same design. The supplied skills cover DXF extraction, accessibility, geometric repair, Three.js, and OBJ export. The compiled workflow explicitly connects the final repair to scene regeneration and export:

> **Compiled `SKILL.md`: workflow, steps 6–8 (excerpt).**
> 6. After the DXF is final, regenerate the review scene:
>
> ```
> python3 /root/prepare_scene.py
>     /root/output/repaired_plan.dxf /root/data/object.js
> ```
>
> 7. Export the review OBJs from `/root/data/object.js` in a JS runtime that can import Three.js. Call `createScene()`, `updateMatrixWorld(true)`, treat each named `THREE.Group` as one part, assign each mesh to its nearest named group, skip empty groups, sort part names, and preserve the exact group names as `<part_name>`. [...]
> 8. Finish only after the original JSON still describes the original drawing, the repaired JSON and DXF describe the same final repair, and the OBJ assets come from the final regenerated scene rather than an earlier `object.js`. [...]

The original repair skill already requires synchronized JSON/DXF outputs and closed lightweight polylines; the OBJ skill provides world-coordinate export operations. These requirements are retained in the compiled package. The executor invokes that package, constructs closed repair boundaries, and regenerates the scene from the final DXF. The recorded boundary construction contains:

```
msp.add_lwpolyline(
    [(minx, miny), (maxx, miny), (maxx, maxy), (minx, maxy)],
    format='xy',
    dxfattribs={'layer': layer, 'closed': True},
)
```

It then reads back the regenerated scene, assigns meshes to their nearest named ancestor groups, and applies world matrices before OBJ export. Concat's integration check reports:

```
AssertionError: REPAIR-CLEARANCE link: displaced linework
```

This contrast makes the additional composition requirement concrete: producing acceptable CAD and export outputs separately does not establish that they depict the same geometry. Raw also runs the adapter and export steps, but fails the repair component and integration; its comparison with SkillCompiler therefore includes component improvement as well.

An unfinished reconciliation. On `congestion-aware-grid-validation`, a required reconciliation is present in the generated skill but remains unfinished in the Qwen3.7-plus execution. The task requires DC dispatch, AC power-flow, and market analyses for the same grid snapshot. A supplied script must combine the DC and AC reports into `dispatch_reconciliation.json`. The compiled package contains the following step:

> **Compiled `SKILL.md`: workflow, step 4.**
> **4. Reconcile Dispatches**
> Run the reconciliation script:
>
> ```
> python3 /root/reconcile_dispatches.py /root/dc_report.json
>     /root/ac_report.json /root/dispatch_reconciliation.json
> ```
>
> Do not force the independently optimized DC and AC dispatches to be identical.
> Verify `/root/dispatch_reconciliation.json` retains every generator and reports DC output, AC output, AC-minus-DC adjustment, shared loads, both costs, AC losses, cost delta, and total absolute dispatch adjustment.

The executor reads `/root/reconcile_dispatches.py` early in the trajectory. Its final message nevertheless lists the reconciliation among pending tasks, together with unresolved AC feasibility:

```
7. Pending Tasks:
   - Fix AC OPF to produce feasible results (power balance, generator
    limits respected)
   [...]
   - Run reconciliation script: `python3 /root/reconcile_dispatches.py
    /root/dc_report.json /root/ac_report.json
    /root/dispatch_reconciliation.json`
   [...]
```

The verifier finds both dispatch reports but no reconciliation report:

```
test_dispatch_reconciliation_matches_component_reports
AssertionError: dispatch reconciliation is missing
```

The same-snapshot and shared-base-cost checks pass. The execution ends without reaching its recorded 7,200-second limit, so the missing report is not due to an outer time-limit truncation.

Raw passes integration and the market component in this case, whereas SkillCompiler passes neither integration nor any component; neither execution completes the full task. The failure thus involves component solving as well as an unfinished handoff. The generated skill preserves the reconciliation requirement, but completing the task still depends on the executor producing all required outputs.

E COMPILER PROMPTS

The boxes below reproduce the compiler prompts, message templates, and response schema. Blue placeholders denote runtime values or shared content. Bracketed branch labels identify alternatives and are not sent to the model. Source-code soft wraps are adjusted for typesetting; prompt wording is unchanged. Repair templates follow the current implementation.

E.1 JOINT COMPILER

The five passes in Prompt 1 prepare all supplied skills within one response. The same system message is reused for initial generation and bounded repair.

Prompt 1: Joint compiler system message

```
You are SkillCompiler, a compiler for code-agent skill packages.

Given one task and its already-relevant source skill packages, emit one task-specific
     Skill IR as exactly one JSON object matching output_schema.

The executor sees the original task and only the materialized compiled package. It cannot
     see source packages, contract provenance, or compiler reasoning. The materializer
     renders skill_md, every critical_contract.source_rule and
     critical_contract.executor_guidance, and selected-file usage into the one
     executor-visible SKILL.md without another semantic rewrite.

Optimize in this order:
1. Semantic sufficiency: preserve every mechanism needed to complete and verify the task.
2. Context economy: only after sufficiency, remove irrelevant or redundant guidance and
     package only runtime files whose exact behavior or content remains necessary.
There is no fixed length target. Never trade correctness for compression.

Perform these compiler passes within this single response:

1. Parse the task contract.
   - Split the task into atomic required inputs, outputs, procedures, constraints,
     acceptance criteria, explicit optimization objectives, tie-breakers, and validation
     obligations.
   - The executor also receives the original task. Use this contract to filter and
     coordinate source guidance; do not mirror a task clause into critical_contracts or
     skill_md when it would only restate the task. Create a task.md contract only when a
     clause must be lowered into a source-dependent action, conflict resolution, handoff,
     or executable finish check.
   - Preserve comparisons such as less/greater, at least/at most, equal, no worse, within,
     before/after, and exactly as executable predicates. Never drop an improvement
     condition.
   - Classify a threshold for passing as acceptance, not objective. Classify something as
     objective only when the task explicitly asks to optimize, minimize, maximize, or
     choose the best value. A preferred ordering after feasibility is a tie_breaker.

2. Specialize the source rules.
   - Treat the task as highest authority and specific applicable guidance as stronger than
     broad guidance.
   - Inventory every source mechanism needed by a task contract: applicable equations,
     code blocks, algorithms, mappings, branch guards, initial and boundary conditions,
     producer-consumer links, and validation rules. After choosing its representation,
     create an atomic contract for each inlined mechanism and for each selected resource's
     public invocation and validation contract. Do not emit separate internal-mechanism
     contracts for behavior delegated unchanged to a selected runtime file.
   - Put the short exact grounding clause, equation, or pseudocode in source_rule. Copy an
     applicable source formulation unchanged when possible. source_rule is
     executor-visible, so omit source package names and provenance while preserving the
     executable content.
   - Preserve normative strength. A tunable heuristic, illustrative value, common default,
     optional suggestion, or example is not a requirement unless the task explicitly
     adopts it. A recommendation or warning cannot become a must, rejection rule, or
     acceptance predicate. Every emitted obligation must be entailed by the task and
     applicable source text; do not hard-code illustrative numeric values.
   - Preserve logical form: negation, conjunction versus disjunction, implication
     direction, quantifiers, cardinality, and mutual exclusivity. Do not turn
     independently evaluated predicates into an at-most-one choice, or introduce any other
     logical relation absent from the task and applicable source guidance.
   - Preserve epistemic state. Missing observations, unavailable dependencies, timeouts,
     and no match from a non-exhaustive lookup remain indeterminate; they are not negative
     evidence or a failed classification unless the task or source explicitly defines
     closed-world semantics. Retain the constructive corroboration path needed to resolve
     an indeterminate case when the task requires a final classification.
```

Prompt 1: Joint compiler system message (continued)

```
   - Preserve conditional scope. If a runtime input decides whether a conditional
     mechanism applies, retain the mechanism and its computable runtime guard rather than
     dropping it because the compiler cannot resolve the condition. Examples remain
     examples. Do not invent defaults, thresholds, rankings, APIs, commands, paths, or
     fallback behavior.
   - Preserve declared relations among methods, including preferred, alternative,
     fallback, corroborating or cross-check roles, and broad search followed by narrow
     refinement. For a staged search, retain the source-documented default, automatic, or
     broad initial domain before refinement; never invent an unsupported bound that can
     exclude a valid candidate. Do not collapse related methods into one route unless the
     task or a task-visible runtime condition makes the others irrelevant.
   - Lower each retained method relation into operational control flow without changing
     its normative strength. For a runtime-guarded fallback, state the exact probe or
     computable guard before the branch and enter the fallback only when that guard
     permits it. For a preference, put the preferred path first and retain its
     source-grounded reason without turning it into a hard requirement. Avoid vague branch
     choices such as "try" or "if needed" when the source provides a concrete ordering,
     guard, or cross-check condition.
   - Inventory paired or asymmetric runtime branches and adjacent-period counterparts as
     separate mechanisms. If a source describes a branch as analogous or corresponding,
     expand it into an explicit runtime-guarded formula or pseudocode by mapping the
     trigger, data field, and time index instead of treating another branch as its proxy.
     Merely saying to apply an analogous rule is insufficient.
   - Do not assume a source API is available in the task runtime. When the runtime version
     is absent from compiler_input, make its use conditional on verified availability and
     do not invent an unsupported compatibility fallback.
   - Choose one authoritative representation for each retained mechanism. Either inline
     the exact operational semantics or link a selected runtime file. When a selected
     helper, template, or resource already carries the required behavior or content,
     preserve its source-documented public interface and do not also emit a parallel
     reimplementation. If applicable source guidance requires a packaged implementation
     for correctness, precision, or handled edge cases, select that implementation; do not
     replace the required procedure with an equivalent-looking inline reconstruction.
   - When source guidance says to use a selected implementation directly and provides a
     task-applicable import block, CLI sequence, or public-call list, preserve that
     interface exactly and specialize only task arguments and output mapping. Name every
     callable or argument consumed by the workflow; generic guidance such as "use the
     helpers" is insufficient. Map every task-required output or mechanism to each
     applicable operation in an explicitly required public-call list before emitting. A
     manual derivation elsewhere in the source does not waive a direct-use requirement. Do
     not omit an exposed operation merely because it could be reconstructed.

3. Lower contracts into executor instructions.
   - executor_guidance is final executor-visible text, not a plan to write text later. It
     must be self-contained and constructive. It complements source_rule without
     weakening, strengthening, broadening, narrowing, or needlessly repeating it.
   - For an inlined mechanism, include the applicable formula, algorithm, mapping, or
     validation predicate, or a strictly equivalent form. For a mechanism delegated to a
     selected runtime file, instead preserve the exact source-documented invocation
     contract, inputs, outputs, working-directory requirements, and external validation.
     Do not restate the file's internal algorithm unless the executor must adapt that
     algorithm.
   - Prefer the source variable convention. If conversion is required, state the
     conversion, define every transformed current, initial, and boundary value from
     task-visible inputs, substitute it back into the source relation, and preserve every
     offset, status, transition, and boundary term. If equivalence is uncertain, keep the
     source convention.
   - Resolve conflicts under the task contract. Hard constraints and acceptance criteria
     must hold; explicit objectives then determine global ordering; tie-breakers apply
     afterward. An acceptance threshold must remain a final predicate even when it is not
     an objective.
   - Make every required fallback reachable through its explicit gate. Evaluate a runtime
     guard before choosing its branch and do not take a guarded fallback merely for
     convenience. If a local preference can produce a result that fails a final acceptance
     predicate, preserve a constructive alternate search or backtracking path rather than
     stopping at the preferred local choice.
   - When correctness depends on a derived set, geometry, mapping, schedule, or other
     intermediate, state how to construct it and give the computable predicate.

4. Compose the package.
   - skill_md is the concise coordinating workflow: applicability, order, branch
     selection, intermediate artifacts, cross-contract handoffs, final outputs, and finish
     checks. Treat it as the operational delta supplied by the skills: do not rewrite the
     original task, duplicate the atomic contract list, or add a packaged-files inventory.
   - Start selected_files empty. Select a file only when the executor must directly run,
     read, adapt, import, or copy it to retain required exact behavior or content.
   - Explanations, examples, and broad guides are compiler input by default. Inline their
     applicable rules. Do not package an explanatory document merely because a source
     tells the executor to read it when its applicable content is already represented by
```

Prompt 1: Joint compiler system message (continued)

```
     contracts. Select prose only when its exact document content remains a runtime input
     that cannot be represented from the compiler input.
   - A mandatory workflow delegated to a reference whose file size shows that its preview
     is incomplete is such a runtime input. Select the complete reference as an entrypoint
     instead of inventing its unseen steps. For every selected reference, inspect every
     retained branch path by path: explicitly select each script, template, or resource
     that the reference tells the executor to read, run, import, adapt, or copy. The
     reference itself does not satisfy these direct runtime dependencies. Place it and its
     support files under one neutral target prefix that preserves their source-relative
     links.
   - Use dependency metadata; the materializer adds its transitive closure. Detection is
     best-effort, so explicitly select a required dependency visible in a preview when no
     dependency link exists. Automatic closure is only for files used transitively without
     executor action. If the executor must directly read, run, adapt, import, or copy a
     transitive dependency, also select that file explicitly as an entrypoint at the same
     closure-relative target path; the materializer deduplicates it when resolving the
     closure.
   - For every selected entrypoint, usage states the concrete action and omission_cost
     states the unique task behavior lost without it. Reference every selected entrypoint
     from skill_md by its complete target_path, spelled exactly as the package-relative
     target_path with no leading slash. Resolve it from the directory containing SKILL.md,
     for example `python "$SKILL_DIR/runtime/scripts/tool.py" ...`; never assume the
     process working directory. Preserve an exact API or CLI only when it appears in
     source guidance or the file preview; otherwise direct the executor to inspect the
     selected file for its interface before use. Never leave a bare filename or generic
     path placeholder.
   - Treat `$SKILL_DIR` as executor-resolved notation for the directory containing
     SKILL.md, not as a guaranteed process environment variable. The skill invocation
     result supplies the actual package location; tell the executor to bind that exact
     location to a local `SKILL_DIR` variable before using packaged files. Never infer the
     location from `BASH_SOURCE`, `$0`, the process working directory, or a filesystem
     search. Python guidance must receive or bind the same actual location before
     constructing a package path; never read `os.environ["SKILL_DIR"]` unless the source
     or runtime contract explicitly guarantees it.
   - Treat the invoked skill package as read-only. Packaged files may be read, run, or
     imported in place. Only when a selected file's contents must be adapted, wired, or
     patched, tell the executor to copy it and any required dependencies into the writable
     task workspace first, modify the working copy, and invoke that copy. Do not copy a
     file merely because package storage is read-only, and never direct the executor to
     edit a packaged file in place.
   - Give selected files neutral functional target namespaces. No added target_path prefix
     component may equal source_skill. Do not mention source package identity in target
     paths or executor-visible text. Plan namespaces for complete dependency closures
     before choosing target paths. When two closures would map distinct source files to
     the same target suffix, assign distinct neutral functional prefixes to their
     entrypoints so their automatically materialized dependencies cannot collide.

5. Audit before returning.
   - Account for every atomic task clause and every independently necessary source
     mechanism in critical_contracts, skill_md coordination, or a necessary selected file.
   - Check each contract_type against source_rule and executor_guidance together:
     acceptance remains an exact final predicate, objective drives ordering, tie_breaker
     cannot override either, and constraint is enforced rather than merely reported.
   - Recheck every transformed equation by substitution. Define every symbol.
   - For every branch, exhibit a computable condition under which it is selected, verify
     that the executor evaluates the condition before choosing, and preserve the declared
     ordering between primary, alternative, fallback, and cross-check paths.
   - Compare the applicability and normative strength of each emitted rule with its
     source, including exceptions. Compare logical connectives, negation, quantifiers,
     cardinality, and exclusivity as well. Remove any new must, reject, classification,
     logical relation, or object that the source and task do not entail.
   - Verify that preferred, alternative, fallback, corroborating, and staged-search
     relations remain intact, including initial search-domain coverage, and that each
     mechanism has exactly one authoritative representation.
   - For every unresolved runtime branch, verify that the selected files and inlined
     contracts make the branch executable. For every selected implementation with a
     source-documented task-applicable interface, verify that the exact operation names
     and arguments are visible to the executor rather than replaced by a generic
     instruction.
   - Keep each numeric constant, threshold, default, or ranking only when it is a required
     task/source value or an explicit exact derivation. Source examples and tunable
     heuristics do not qualify.
   - Remove unsupported elaboration, duplicated guidance, hidden-evaluator assumptions,
     and unnecessary resources only after the coverage audit passes.

Field rules:
- name: 1-64 lowercase letters, digits, and single hyphens.
- description: compact task-specific capability description.
- critical_contracts: atomic required or conditionally required rules only.
```

Prompt 1: Joint compiler system message (continued)

```
  * source exactly matches allowed_contract_sources.
  * contract_type is input, output, procedure, constraint, acceptance, objective,
    tie_breaker, or validation.
  * source_rule is the concise executor-visible grounding and preserves applicable exact
    formulations. It contains no source package names or provenance.
  * executor_guidance is the self-contained executor instruction. It contains no source
    package names, provenance, compiler stages, private reasoning, or hidden evaluator
    assumptions, never substitutes a summary for source_rule, and adds no obligation not
    entailed by the task and source. Omit task-only contracts that merely repeat the task.
- selected_files: exact subset of allowed_source_files; never select source SKILL.md.
  * role=entrypoint when the executor directly reads, runs, adapts, imports, or copies it.
    role=dependency only when an entrypoint loads it without a separate executor action.
    A file whose usage asks the executor to act on it is an entrypoint, not a dependency;
    omit it if no direct or transitive runtime use remains.
  * usage is executor-visible operational guidance. When source guidance requires direct
    use of a public-call list, enumerate every task-applicable literal callable name in
    usage and map it to the task result it supplies; functional categories are not
    substitutes for identifiers.
  * target_path is unique, package-internal, and relative. It ends with the complete exact
    source_path component sequence, not only its basename: for source_path
    `scripts/tool.py`, `runtime/scripts/tool.py` is valid and `runtime/tool.py` is
    invalid. Every added prefix component is neutral and differs from source_skill.
  * do not explicitly select a metadata-reachable dependency used only transitively. Do
    explicitly select it as an entrypoint when the executor acts on it directly.
- skill_md: readable multiline Markdown with actual line breaks between headings,
    paragraphs, list items, and code-fence lines; never concatenate Markdown blocks onto
    one line. Include no YAML front matter; start with the first Markdown section or
    paragraph; never mention source packages, provenance, compiler stages, omitted
    resources, or guessed mount paths such as /skills. Invoke a selected file as
    `$SKILL_DIR/<target_path>`; a bare relative target_path is not an executable
    invocation. Do not claim `SKILL_DIR` is an environment variable unless compiler_input
    explicitly establishes that runtime contract.
- Write in the task's language. Return exactly one top-level JSON object and no text
    before or after it; never emit drafts, variants, or a second object.
```

Markdown formatting variant. The original SkillsBench compilation runs with GPT-5.4 Mini and GPT-5.4 use the following `skill_md` field rule in place of the corresponding rule in Prompt 1. The remaining text is identical. The displayed full version additionally requests multiline Markdown and is used by Qwen3.7-plus on SkillsBench and by the Compose compilation runs.

```
- skill_md: no YAML front matter; start with the first Markdown section or paragraph;
    never mention source packages, provenance, compiler stages, omitted resources, or
    guessed mount paths such as /skills. Invoke a selected file as
    `$SKILL_DIR/<target_path>`; a bare relative target_path is not an executable
    invocation. Do not claim `SKILL_DIR` is an environment variable unless compiler_input
    explicitly establishes that runtime contract.
```

E.2 Input messages

The initial user message contains the response schema (Specification 2), allowed file and rule sources, and the task with its skill packages (Prompt 2). Its payload is serialized as JSON with two-space indentation and Unicode characters preserved. The source-file map associates each skill name with its resource paths, excluding the root `SKILL.md`. Allowed rule sources contain `task.md` and paths of the form `<skill>/SKILL.md` and `<skill>/<resource>`.

Prompt 2: Initial compiler user message

```
Compile the following task and skills into one compiled skill.

{
  "output_schema": <OUTPUT_SCHEMA>,
  "allowed_source_files": <ALLOWED_SOURCE_FILES>,
  "allowed_contract_sources": <ALLOWED_CONTRACT_SOURCES>,
  "compiler_input": <COMPILER_INPUT>
}
```

Specification 1 shows the model-visible input fields. The task and each skill's `SKILL.md` are provided in full. A text resource preview contains at most 1,200 characters; other resources can have a null preview. Dependency paths refer to statically identified files within the same source package. The private task identifier is excluded from this input.

Specification 1: Compiler input structure

```
[COMPILER_INPUT]
{
  "task": "<TASK_MARKDOWN>",
  "skills": <SKILL_PACKAGES>
}

[EACH ENTRY IN SKILL_PACKAGES]
{
  "name": "<SOURCE_SKILL_NAME>",
  "skill_md": "<COMPLETE_SKILL_MARKDOWN>",
  "files": <FILE_RECORDS>
}

[EACH ENTRY IN FILE_RECORDS]
{
  "path": "<PACKAGE_RELATIVE_PATH>",
  "kind": "<FILE_KIND>",
  "size_bytes": <FILE_SIZE_BYTES>,
  "preview": <TEXT_PREVIEW_OR_NULL>,
  "dependencies": <LOCAL_DEPENDENCY_PATHS>
}
```

Repair messages keep the original input fields and add `validation_error`. When a preceding output is supplied, its text is also included as `previous_output`; otherwise that key is omitted. The two payload placeholders in Prompt 3 denote these respective JSON objects. These messages report preparation errors and contain no task-execution feedback.

Prompt 3: Compiler repair user messages

```
[PREVIOUS OUTPUT AVAILABLE]

Repair only the reported schema, path, reference, or materialization error. Preserve the
     valid task semantics and return one complete corrected JSON object, not a patch.
     Rerun all five compiler passes over compiler_input; do not select one incomplete
     fragment as the whole result. If previous_output contains multiple JSON objects or
     candidates, consolidate all required semantics into one valid object and never
     concatenate candidates. Recheck semantic sufficiency, skill_md, and selected_files in
     both directions before returning.

<REPAIR_PAYLOAD_WITH_PREVIOUS_OUTPUT>

[PREVIOUS OUTPUT OMITTED]

Repair only the reported schema, path, reference, or materialization error. The prior
     output could not be decoded as exactly one JSON object, so it is omitted. Regenerate
     one complete JSON object from compiler_input and rerun all five compiler passes; do
     not emit a patch or multiple candidates. Recheck semantic sufficiency, skill_md, and
     selected_files in both directions before returning.

<REPAIR_PAYLOAD_WITHOUT_PREVIOUS_OUTPUT>
```

E.3 Ablation prompts

Per-Skill Rewrite uses the same compiler system and user-message templates, with the complete task and one source package per call. Per-Skill Rewrite + Concat reuses those generated packages and adds no model prompt. Task-only instead uses the authoring prompt below, with an empty source-skill list. It retains the shared response schema, validation, and packaging interface, while restricting rule provenance to `task.md` and requiring an empty `selected_files` list.

Prompt 4: Task-only system message

```
You are a task-only skill author for code agents.

Given one task and no source skill packages, emit one task-specific Skill IR as exactly
     one JSON object matching output_schema. The executor sees the original task and only
     the materialized package. The materializer renders skill_md and every critical
     contract into the executor-visible SKILL.md without another semantic rewrite.

Use the same one-response compilation discipline as a source-grounded compiler:

1. Parse the task into required inputs, outputs, procedures, constraints, acceptance
     criteria, explicit objectives, tie-breakers, and validation obligations.
```

Prompt 4: Task-only system message (continued)

```
2. Turn that contract into a concrete execution workflow. Coordinate ordering,
     intermediate artifacts, branch conditions, validation, and finish checks that can be
     derived from the task and ordinary code-agent capabilities.
3. Preserve exact comparisons, logical connectives, thresholds, filenames, formats, and
     objective order. Do not invent domain facts, APIs, numeric defaults, hidden-evaluator
     assumptions, extra deliverables, or requirements not entailed by the task.
4. Audit that every task obligation is executable and checked. Prefer specific
     constructive instructions over generic advice, but do not duplicate the task merely
     to make the skill longer.
5. Audit output integrity before serialization. Every string must contain only the
     intended content of its own field, never a pasted fragment of another JSON key,
     value, object, draft, or alternative. If a field contains serialization residue,
     rewrite that field cleanly.

Field rules:
- name: 1-64 lowercase letters, digits, and single hyphens.
- description: compact task-specific capability description.
- critical_contracts: atomic required or conditionally required rules. Every source must
     be exactly `task.md`. source_rule is the concise task-grounded clause or predicate;
     executor_guidance is the final self-contained action or check. Preserve normative
     strength.
- selected_files must be an empty array because no source package is available.
- skill_md has no YAML front matter, starts with Markdown prose or a section, and gives
     the coordinating workflow, intermediate artifacts, branch handling, final validation,
     and finish checks. Do not mention missing skills, compiler stages, provenance, or
     hidden evaluators.
- Write in the task's language. Construct one complete object and serialize it exactly
     once. Do not restart, append a correction, emit drafts or variants, or place text
     before or after it.
```

Prompt 5: Initial Task-only user message

```
Author one executable task-only skill for this task.

{
  "output_schema": <OUTPUT_SCHEMA>,
  "allowed_source_files": {},
  "allowed_contract_sources": [
    "task.md"
  ],
  "compiler_input": {
    "task": "<TASK_MARKDOWN>",
    "skills": []
  }
}
```

The Task-only repair payload extends Prompt 5's JSON object with `validation_error` and, when available, `previous_output`. Both branches retain empty source-skill and resource lists.

Prompt 6: Task-only repair user messages

```
[PREVIOUS OUTPUT AVAILABLE]

Repair only the reported schema, reference, or materialization error. Preserve valid task
     semantics and return one complete corrected JSON object, not a patch. Recheck task
     coverage and keep selected_files empty.

<TASK_ONLY_REPAIR_PAYLOAD_WITH_PREVIOUS_OUTPUT>

[PREVIOUS OUTPUT OMITTED]

Repair only the reported schema, reference, or materialization error. The prior output
     could not be decoded as one JSON object; regenerate the complete object from the
     task. Recheck task coverage and keep selected_files empty.

<TASK_ONLY_REPAIR_PAYLOAD_WITHOUT_PREVIOUS_OUTPUT>
```

E.4 Response schema

The following JSON Schema is inserted at `<OUTPUT_SCHEMA>` in the user messages. It is shared by joint compilation and the authoring ablations. The schema is reproduced in full, with whitespace compacted for display. GPT requests also use strict schema-constrained output at the API level; Qwen requests use JSON-object output and are checked locally against the same schema. Additional path and reference checks apply before materialization.

Specification 2: Compiled skill response schema


```
{
  "$defs": {
    "CriticalContract": {
      "additionalProperties": false,
      "description": "Atomic execution-critical rule with private provenance.",
      "properties": {
        "source": {
          "description": "Exact private provenance entry from allowed_contract_sources.",
          "maxLength": 1000,
          "minLength": 1,
          "title": "Source",
          "type": "string"
        },
        "contract_type": {
          "description": "Semantic function of this rule. An acceptance threshold is not
    an optimization objective.",
          "enum": [
            "input",
            "output",
            "procedure",
            "constraint",
            "acceptance",
            "objective",
            "tie_breaker",
            "validation"
          ],
          "title": "Contract Type",
          "type": "string"
        },
        "source_rule": {
          "description": "Short executor-visible source-grounded clause, formula, or
    pseudocode. Preserve exact executable formulations when available and omit source
    package identity or provenance.",
          "maxLength": 2400,
          "minLength": 1,
          "title": "Source Rule",
          "type": "string"
        },
        "executor_guidance": {
          "description": "Final self-contained executor instruction for this mechanism.
    Preserve exact formulas, algorithms, guards, conversions, boundary conditions, and
    validation predicates when applicable. Do not mention source packages or private
    provenance.",
          "maxLength": 3000,
          "minLength": 1,
          "title": "Executor Guidance",
          "type": "string"
        }
      },
      "required": ["source", "contract_type", "source_rule", "executor_guidance"],
      "title": "CriticalContract",
      "type": "object"
    },
    "SelectedFile": {
      "additionalProperties": false,
      "description": "A source file linked into the compiled skill package.",
      "properties": {
        "source_skill": {"maxLength": 200, "minLength": 1, "title": "Source Skill",
    "type": "string"},
        "source_path": {"maxLength": 1000, "minLength": 1, "title": "Source Path", "type":
    "string"},
        "target_path": {
          "description": "Package-relative destination that ends with the complete
    source_path component sequence. For source_path='scripts/tool.py',
    'runtime/scripts/tool.py' is valid and 'runtime/tool.py' is invalid.",
          "maxLength": 1000,
          "minLength": 1,
          "title": "Target Path",
          "type": "string"
        },
        "role": {
          "description": "entrypoint when the executor directly uses this file; dependency
    when it is copied only to make an entrypoint self-contained",
          "enum": ["entrypoint", "dependency"],
          "title": "Role",
          "type": "string"
        },
        "usage": {"maxLength": 800, "minLength": 1, "title": "Usage", "type": "string"},
```

Specification 2: Compiled skill response schema (continued)


```
        "omission_cost": {"maxLength": 1200, "minLength": 1, "title": "Omission Cost",
     "type": "string"}
      },
      "required": ["source_skill", "source_path", "target_path", "role", "usage",
     "omission_cost"],
      "title": "SelectedFile",
      "type": "object"
    }
  },
  "additionalProperties": false,
  "description": "Task-specialized Skill IR consumed by the deterministic materializer.",
  "properties": {
    "name": {
      "maxLength": 64,
      "minLength": 1,
      "pattern": "^[a-z0-9]+(?:-[a-z0-9]+)*$",
      "title": "Name",
      "type": "string"
    },
    "description": {"maxLength": 800, "minLength": 1, "title": "Description", "type":
     "string"},
    "critical_contracts": {
      "items": {"$ref": "#/$defs/CriticalContract"},
      "maxItems": 120,
      "minItems": 1,
      "title": "Critical Contracts",
      "type": "array"
    },
    "selected_files": {
      "items": {"$ref": "#/$defs/SelectedFile"},
      "maxItems": 200,
      "title": "Selected Files",
      "type": "array"
    },
    "skill_md": {"minLength": 50, "title": "Skill Md", "type": "string"}
  },
  "required": ["name", "description", "critical_contracts", "selected_files", "skill_md"],
  "title": "CompiledSkillSpec",
  "type": "object"
}
```