Generative Skill Composition for LLM Agents

Xinyu Zhao [1], **Zhen Tan** [2], **Vaishnav Tadiparthi** [3], **Nakul Agarwal** [3], **Kwonjoon Lee** [3], **Ehsan Moradi Pari** [3], **Hossein Nourkhiz Mahjoub** [3] **and Tianlong Chen** [1] ✉

[1] University of North Carolina at Chapel Hill [2] Arizona State University [3] Honda Research Institute USA

Abstract

Recent LLM agents benefit from skills for solving complex tasks. Skills encapsulate modular packages of procedural knowledge and instructions for performing specialized tasks, such as setting up a sandboxed environment, running a test suite, or refactoring a function across multiple files. As skill libraries grow and become reusable across tasks and domains, selecting an appropriate skill composition has emerged as a central bottleneck. Existing approaches fall into two categories. One exposes the agent's reasoning to the entire skill collection; the other performs skill retrieval via embeddings or LLM-based rerankers. Both provide useful insights; however, they miss the structural nature of skill composition, which is a joint decision over which skills, how many, and in what order—three dimensions that cannot be decoupled. We formalize this as structured skill composition: given a task and a skill library, predict an executable skill plan that jointly specifies the activated subset, count, and execution order. We propose `SkillComposer`, which instantiates structured skill composition as task-conditioned skill sequence prediction. `SkillComposer` uses a constrained autoregressive decoder over skill identifiers, so subset, count, and order emerge jointly from a single decoding pass, and dependencies between successive skills are captured naturally. We build a training set of task–composition pairs from a real, human-curated skill library. We then evaluate `SkillComposer` along two axes: composition quality on a held-out test set, and downstream task success on SkillsBench across two production-grade coding agents. On {GPT-5.2-Codex, Gemini-3-Pro-Preview}, `SkillComposer` raises the pass rate by {$+23.1$, $+18.2$} pp over the no-skill baseline, surpassing top-3 retrieval and matching the gold-skill retrieval upper bound at lower prompt-token cost.

Project Page: https://skill-composer.github.io/

1 Introduction

Skill libraries allow LLM agents to apply procedural knowledge across complex tasks such as structured document generation, software development, and computer-use automation (Jimenez et al., 2024; Yang et al., 2024; Xie et al., 2024; Zhou et al., 2024; Trivedi et al., 2024). A skill is a modular, reusable unit of procedural knowledge, *e.g.* bundling natural-language instructions, scripts, and supporting resources that an agent dynamically loads into its working context to perform a specialized subtask, such as coordinating code review and test execution, generating artifacts, or producing structured documents (Anthropic, 2025; Xu & Yan, 2026; Jiang et al., 2026). The research community has made substantial progress on curating skills for varied tasks (Wang et al., 2023; 2026; Xia et al., 2026; Li et al., 2025; Jiao et al., 2026; Yue et al., 2025; Liu et al., 2026a), driving skill libraries to grow rapidly in size. This shifts the inference-time bottleneck from obtaining skills to composing the right collection of skills: to improve

✉ Corresponding authors

Preprint. Under review.

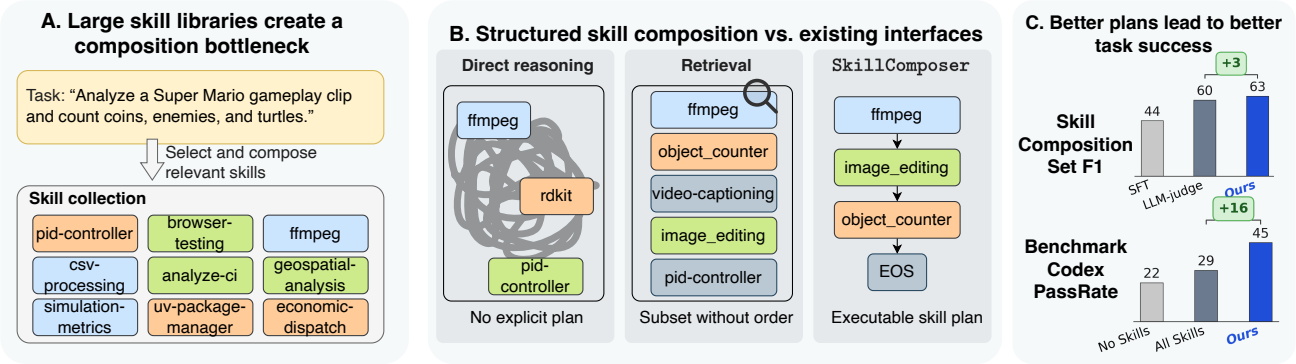


Figure 1: **Structured skill composition with `SkillComposer`. (A)** Large skill libraries create a composition bottleneck. To solve complex tasks, an agent must decide not only which skills to use, but also their exact count and execution order. **(B)** Existing paradigms: directly exposing the agent to all skill options leaves composition implicit within an unstructured execution trace, while retrieval methods only return an unordered subset of candidates. We propose `SkillComposer`, which explicitly predicts an ordered, executable skill sequence. **(C)** By structuring the composition process, `SkillComposer` improves both plan exact match and downstream task success rates on SkillsBench.

task performance, the agent must decide which skills to load, how many, and in what order they should be used.

Current approaches for selective skill use fall into two paradigms. Retrieval ranks skills independently using LLM-as-a-judge or task-skill embedding similarity (Zheng et al., 2026; Su et al., 2026; Li et al., 2025), while end-to-end planning exposes the agent to the full skill library, where the agent needs to reason to trigger appropriate skills at the same time as approaching input tasks (Wang et al., 2023; Yao et al., 2023; Schick et al., 2023; Xia et al., 2026). These interfaces are sufficient when a task maps cleanly to a single dominant skill, but they leave important structural aspects implicit in compositional tasks. For example, consider a request to "locate a deprecated API call, refactor it across the codebase, and run the regression suite." A useful plan should first identify the relevant call sites, then apply the refactor, and finally run tests to validate the change. Retrieval can surface individually relevant skills for search, editing, and testing, but a ranked list alone does not specify how many skills should be used or the order in which they should be executed.

We formulate structured skill composition as task-conditioned skill sequence prediction. Given a natural-language task and a fixed reusable skill library, the model predicts an ordered sequence of skill identifiers ending with a stop symbol. We propose `SkillComposer`, a generative framework that uses a constrained autoregressive decoder over existing library skills, where subset selection, skill-count prediction, and skill ordering emerge jointly from a single decoding pass. Because each prediction is conditioned on the task, the skill library, and the previously selected skills, `SkillComposer` can capture dependencies between successive skills without requiring an explicit execution order at inference time. Also, the output vocabulary is a valid skill identifier, the predicted plan is inspectable, and can be loaded directly into downstream agents.

To build `SkillComposer`, we construct a dataset of task-skill composition pairs grounded in a real, human-curated skill library. Starting from real task-skill composition seeds from SkillBench (Li et al., 2026), we build a skill dependency graph using skill metadata and observed workflow co-occurrence. We then use layered synthesis and filtering to obtain supervision for both single-skill grounding and multi-skill compositions. This construction is designed to cover long-tail individual skills and dependency-aware skill chains, while keeping the output space tied to executable skills in the library.

We evaluate `SkillComposer` along two complementary axes. First, we measure composition quality on evaluation sets from both synthetic and held-out real data, testing whether the predicted plan matches the target skill subset, count, and order. Second, we measure downstream task success on SkillsBench across two production-grade coding agents (GPT-5.2-Codex, Gemini-3-Pro-Preview) to test whether better skill plans translate into better agent execution. Our main contributions are:

- ⋆ We formalize inference-time skill use for LLM agents as a structured prediction problem over a fixed skill library, where the output plan jointly determines which skills to activate, how many skills are needed, and in what order they should be executed.
- ⋆ We construct a dataset grounded in a real, human-curated skill library. Starting from real task-composition seeds, we build a skill dependency graph and use layered synthesis with quality filtering to obtain supervision for both single-skill and multi-skill dependency-aware composition.
- ⋆ We propose `SkillComposer`, a task-conditioned skill sequence predictor with a constrained autoregressive decoder over skill identifiers. `SkillComposer` unifies subset selection, cardinality prediction, and ordering into a single decoding process while ensuring that every generated element corresponds to an executable library skill.
- ⋆ We evaluate `SkillComposer` on composition quality and downstream task success across two production-grade coding agents. On {Codex, Gemini}, `SkillComposer` raises SkillsBench pass rate by $\{+23.1, +18.2\}$ pp over the no-skill baseline, surpassing retrieval and matching the gold-skill retrieval upper bound at lower prompt-token cost.

2 Related Works

Skill libraries and discovery. A growing line of work equips LLM agents with reusable skills or tools that can be retrieved at deployment time (Wang et al., 2023; Yuan et al., 2023; Qian et al., 2023; Ma et al., 2025). While these methods have shown clear benefits of maintaining a skill inventory, they all adopt flat retrieval at selection time, ranking skills independently by embedding similarity without considering how many skills are needed or how they depend on one another. In contrast, our work takes a curated skill library as given and addresses the complementary problem of structured composition: jointly predicting the skill subset, the number of skills, and execution order. Recent work further validates the growing importance of skill-based agent design across retrieve-and-rerank routing, retrieval-augmented skill use, benchmarking, lifecycle taxonomies, RL-based skill construction, and analyses under realistic retrieval conditions (Zheng et al., 2026; Li et al., 2025; Liu et al., 2026a; Su et al., 2026; Li et al., 2026; Jiang et al., 2026; Xia et al., 2026; Liu et al., 2026b). None of these efforts model skill selection as closed-vocabulary sequence generation with explicit cardinality and ordering decisions.

Tool-level planning and composition. A parallel research thread plans over tool or API action spaces at the atomic function-call level, casting selection and ordering as decomposition, search, graph evaluation, vocabulary embedding, or graph-structured planning (Shen et al., 2023; Zhuang et al., 2023; Shen et al., 2024; Hao et al., 2023; Wu et al., 2024). Although these systems also reason about inter-step dependencies, they operate at the API-call level, where typed signatures and return values provide strong structural signals for selection and ordering. Our work operates at a coarser skill level, where each skill encapsulates a reusable multi-step procedure spanning several API calls (e.g., "schema grounding" or "query reformulation"), and this shift introduces challenges API-level methods do not face: skills lack typed signatures, so dependencies are latent and task-logical rather than type-induced, and the catalog is small but highly interacting, swapping or reordering even two skills can flip task outcome. Flat similarity retrieval and atomic-action search are thus both ill-suited, motivating skill composition as joint prediction.

3 Preliminaries

Following the open Agent Skills standard [1] and (Jiang et al., 2026), an agentic skill is a reusable procedural module that augments agent behavior at inference time by injecting procedural knowledge into the prompt context, without modifying model weights. Formally:

Definition 3.1 (Skills). We define each skill as $s_i = (m_i, C_i, \pi_i, T_i, R_i)$, where m_i is metadata (name + one-line description, e.g. "*flood-detection – compare water levels to thresholds; count flood days*), C_i is an applicability condition (e.g."*input contains a time-indexed water-level series and station thresholds*"), π_i is a procedural policy (e.g."*aggregate instantaneous values to daily extremes, then compare against the action/minor/moderate/major bands*"), T_i is a termination condition (e.g., "*a per-station flood_days count has been written to the requested output path*")), and R_i is an optional callable interface or

[1] https://agentskills.io/what-are-skills

Input: task x, environment c, skill library $\mathcal{S} = \{s_1, \ldots, s_K\}$ with $K = 196$.

`[Skill library - name + one-line description per entry]`
s_{55} `flood-detection` – compare water levels to thresholds; count flood days.
s_{104} `nws-flood-thresholds` – fetch action/minor/moderate/major flood stages from NWS.
s_{184} `usgs-data-download` – pull USGS streamflow / gage-height series.
... 193 other skills, e.g. `pdf`, `rdkit`, `pptx`, ...

`[Task` x`]` "Find Michigan USGS stations that experienced flooding during April 1–7, 2025; write (`station_id`, `flood_days`) to `flood_results.csv`."

`[Environment` c`]` `/root/data/michigan_stations.txt` (streamflow records).

Predicted sequence: $(s_{104}, s_{184}, s_{55}) \rightarrow$ (`nws-flood-thresholds`, `usgs-data-download`, `flood-detection`)

Figure 2: Example of selecting an ordered skill sequence from a large skill library given a task and environment.

supporting resource (e.g."*a Python helper, a REST endpoint such as the USGS dataretrieval API, or a bundled lookup table*").

Unlike tools (atomic API calls), plans (ephemeral task-specific reasoning), and prompt templates (static text without applicability gating), a skill encodes reusable procedural knowledge that persists across tasks and sessions. We further define a skill library as follows:

Definition 3.2 (Skill Library). Let $\mathcal{S} = (s_1, \ldots, s_K)$ denote a library of K reusable skills. Each skill s_i exposes metadata m_i for runtime discovery via progressive disclosure: agents load compact metadata at startup and activate full instructions on demand.

We treat $\mathcal{S}$ as fixed at training time because real agentic deployments ship with curated skill packs rather than open-ended skill generation (Li et al., 2026), thereby isolating the composition problem from the orthogonal skill-creation problem. With these, we can formulate our problem:

Definition 3.3 (Task-conditioned Skill Sequence Prediction). Given a task description $x \in \mathcal{X}$, an environment context $c \in \mathcal{C}$, and a skill library $\mathcal{S} = \{s_1, \ldots, s_K\}$, we formulate skill composition as task-conditioned skill sequence prediction. A parameterized model f_θ predicts a variable-length sequence of skill indices:

$$\widehat{\mathbf{z}} = (\widehat{z}_1, \widehat{z}_2, \ldots, \widehat{z}_{\widehat{n}}, \texttt{STOP}) = f_\theta(x, c, \mathcal{S}) \tag{3.1}$$

where θ are learnable parameters, each $\widehat{z}_t \in \{1, \ldots, K\}$ indexes a skill in $\mathcal{S}$, and `STOP` is a special end-of-sequence symbol that signals termination. The predicted skill count $\widehat{n}$ is determined by the position of `STOP`. The predicted skill sequence is then recovered by index lookup: $\widehat{\mathbf{s}} = (s_{\widehat{z}_1}, s_{\widehat{z}_2}, \ldots, s_{\widehat{z}_{\widehat{n}}})$.

The prediction jointly resolves three coupled aspects: *(1) subset selection* — which skills are relevant; *(2) skill count* — how many skills are needed (task-dependent, not fixed or predefined); and *(3) ordering* — in what sequence the selected skills should be composed. At inference time, the resolved skills $\widehat{\mathbf{s}}$ are loaded into the agent's context in the predicted order, providing procedural guidance for downstream execution. Figure 2 illustrates a concrete instance: the model receives a task with the skill library (each entry as a compact *name + description*), and outputs a short index sequence ending in `STOP`, which is resolved to full skill instructions before being prepended to the agent's prompt.

4 `SkillComposer`: Generative Composition of Skill Sequences

4.1 Overview

Inspired by recent work that frames retrieval and tool selection as generation over a closed vocabulary (Rajput et al., 2023; Hao et al., 2023), we treat each library index $i \in \{1, \ldots, K\}$ as a primitive output token of a small task-conditioned decoder, with `STOP` marking sequence termination. This single sequence model jointly resolves the three coupled aspects identified in §3: *which* skills (subset selection), *how many* (skill count), and *in what order* (procedural ordering). Taking

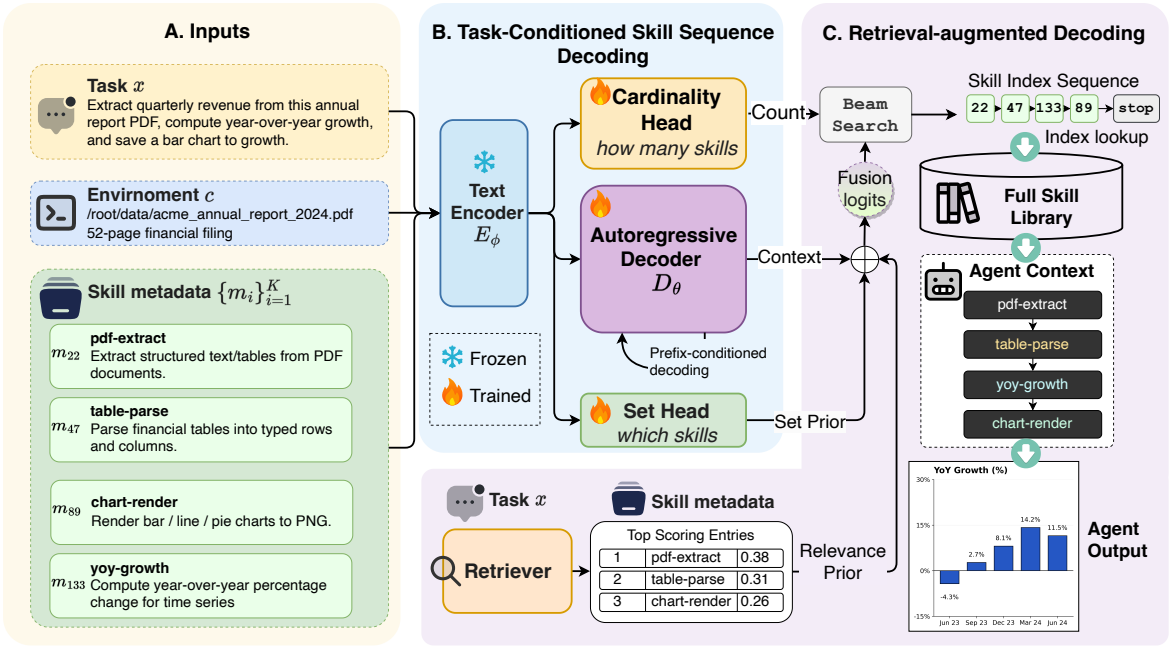


Figure 3: `SkillComposer` method overview. **(A)** Given a task, environment context, and compact metadata from a fixed skill library, `SkillComposer` encodes the task–library context and predicts a variable-length ordered sequence of skill indexes. **(B)** The autoregressive decoder produces contextual skill logits, while auxiliary cardinality and set heads estimate how many skills are needed and which skills are relevant. **(C)** During retrieval-augmented decoding, contextual logits are fused with retrieval and set-membership priors, and the predicted count guides length / STOP control. The resulting skill-index sequence is resolved into full skill packages and loaded into the downstream agent context in the predicted order.

the flood-analysis task in Figure 2 as an example, the skill composing model reads the task x, the environment c (the Michigan stations file), and the compact metadata $\{m_i\}_{i=1}^K$, and emits the index sequence $(104, 184, 55, \texttt{STOP})$. Index lookup recovers the ordered plan $\widehat{\mathbf{s}} = (s_{104}, s_{184}, s_{55}) = (\texttt{nws-flood-thresholds}, \texttt{usgs-data-download}, \texttt{flood-detection})$.

`SkillComposer` has three components, illustrated in Figure 3: (*i*) a frozen text encoder that maps the serialized prompt $P(x, c, \mathcal{S})$ into a dense task representation; (*ii*) a constrained autoregressive decoder with auxiliary cardinality and set-membership heads; and (*iii*) a retrieval-augmented decoding step that fuses a lexical similarity prior into the skill logits. We describe each in turn.

4.2 Task-Conditioned Composer

Skill tokenization. We start with tokenizing each library skill s_i by its integer index $i \in \{1, \ldots, K\}$, yielding the output vocabulary $\mathcal{V} = \{1, \ldots, K\} \cup \{\texttt{STOP}\}$. Only the compact metadata m_i is consumed during prediction; the full procedural policy π_i and resources R_i are activated only after $\widehat{\mathbf{s}}$ is resolved, mirroring progressive disclosure in skill libraries.

Task Encoder. A frozen pretrained text encoder E_ϕ maps the serialized prompt $P(x, c, \mathcal{S})$ into a pooled task vector $\mathbf{h} = W_{\text{proj}}\mathbf{h}_x \in \mathbb{R}^d$. We instantiate E_ϕ as Qwen3-Embedding-0.6B (last-token pooled, output dimension 1024) and project to $d = 256$. The encoder parameters ϕ are frozen. The projection W_{proj}, decoder parameters θ, and auxiliary-head parameters ψ, ξ are trained.

Autoregressive decoder. A small transformer decoder D_θ predicts each token conditioned on the projected task vector and all previously generated tokens $\widehat{z}_{<t} := (\widehat{z}_1, \ldots, \widehat{z}_{t-1})$:

$$p_\theta(\mathbf{z} \mid x, c, \mathcal{S}) = \prod_{t=1}^{n+1} p_\theta\left(z_t \mid \mathbf{h}, \mathbf{z}_{<t}\right) \tag{4.1}$$

We instantiate D_θ as a 3-layer, 256-dim transformer with 4 attention heads. Skills s_i are surfaced to the decoder via cross-attention to projected metadata embeddings, so that the decoder distinguishes the K candidates by their natural-language descriptions rather than their indexes alone.

Factorized supervision via auxiliary heads. The autoregressive head models the joint distribution over the three aspects of skill composition identified in §4.1: *which* skills, *how many*, and *in what order*. This joint formulation is desirable for ordering, where conditioning on $\widehat{z}_{<t}$ is essential, but it dilutes the supervision available to the other two aspects: the entire length signal is carried by a single `STOP` position, and the positive supervision for a relevant skill appears only at its gold position. The model receives no order-agnostic signal that the skill is relevant, independent of where it appears. We therefore complement the AR head with two task-conditional auxiliary heads, one per remaining aspect, each attached to the task vector $\mathbf{h}_x$ and trained jointly with the sequence loss. The AR head retains responsibility for ordering, while cardinality and set membership receive dedicated supervision channels and can also be reused as decoding priors at inference time.

Cardinality head (how many). A linear classifier on $\mathbf{h}_x$ predicts the skill count $\widehat{n} \in \{1, \ldots, N_{\max}\}$ directly (we set $N_{\max}$=8),

$$p_\psi(\widehat{n} \mid x, c) \;=\; \texttt{softmax}(W_n\, \mathbf{h}) \tag{4.2}$$

yielding a length signal that is independent of the AR head's emission of `STOP` and can be used to softly bias or hard-clip decoding.

Set head (which skills). A pairwise matcher g_ξ scores each library skill s_i independently against the task vector,

$$\sigma_i = g_\xi(\mathbf{h}, \mathbf{e}_i) = \texttt{MLP}_\xi([\mathbf{h}; \mathbf{e}_i; \mathbf{h} \odot \mathbf{e}_i; |\mathbf{h} - \mathbf{e}_i|]) \tag{4.3}$$

where $\mathbf{e}_i = W_m E_\phi(m_i) \in \mathbb{R}^d$ is the projected metadata embedding of s_i and the four concatenated terms capture identity, interaction, and distance between task and skill. $\texttt{MLP}_\xi$ is a 2-layer MLP with hidden width 256 and a single output logit. Supervision is binary cross-entropy against the gold membership indicator $\mathbb{1}[s_i \in \widehat{\mathbf{s}}]$, so every relevant skill receives a direct gradient independent of its position in $\mathbf{z}$.

4.3 Retrieval-Augmented Decoding

The autoregressive decoder produces a contextual prediction that conditions every step on the previously decoded output, routing all task information through the dense vector $\mathbf{h}_x$ and learned skill embeddings. Beyond this contextual channel, two structural properties of skills motivate an additional inference-time prior. First, the skill library is heavy-tailed: many skills appear in only one or two training tasks (§5.1). Learned representations have a weak signal for discriminating them, whereas retrieval scores draw on the full library corpus and require no per-skill training data. Second, the skill vocabulary structure makes such an inference-time prior essentially free: each output index corresponds to a fixed metadata document, so any task-skill relevance scorer can be precomputed once per task and reused across all decoding steps without modifying the decoder. We therefore complement the contextual channel with a second, position-independent channel that scores the standalone semantic relevance of each library skill to the task: how well skill s_i matches (x, c) on its own merits. Together, the decoder captures contextual ordering, while a retrieval prior contributes task-skill semantic evidence that the contextual channel is not specialized to express.

Task-skill relevance prior. For each library skill s_i, let $r(x, s_i) \in \mathbb{R}$ denote a task–skill relevance score produced by an off-the-shelf retriever r applied to the task text as query and the skill metadata as documents. We instantiate r as TF–IDF cosine similarity over a unigram–bigram vocabulary built from the library; an ablation comparing this choice to BM25 and dense Qwen3-Embedding cosine is reported in §5.5. The scores $\{r(x, s_i)\}_{i=1}^K$ are precomputed once per task and reused at every decoding step at negligible cost, since each output index corresponds to fixed skill metadata.

Logit fusion at decoding time. At each decoding step t, the raw decoder logit $\ell_t(i)$ for skill index $i \in \{1, \dots, K\}$ is replaced by the fused logit, where $\bar{r}_i$ is a normalized or calibrated retrieval score.

$$\underbrace{\widetilde{\ell}_t(i)}_{\text{fused logit}} = \underbrace{\ell_t(i)}_{\text{contextual}} + \alpha \cdot \underbrace{\bar{r}_i}_{\text{relevance}} + \beta \cdot \underbrace{\sigma_i}_{\text{set}}, \qquad i \in \{1, \dots, K\}, \tag{4.4}$$

with fixed scalars $\alpha, \beta \geq 0$ tuned on validation Set F1 (we use α=1.0, β=0.5); sensitivity to these weights is reported in §5.5, where the surface is bowl-shaped within ± 2 pp Set F1 of the operating point. We do not add retrieval or membership priors directly to the `STOP` logit. Termination is therefore controlled primarily by the AR stop logit and, when used, the cardinality prior. The fused logits $\widetilde{\ell}_t$ are then passed to softmax and beam search as usual.

In sum, `SkillComposer` yields three interpretable signals at every decoding step: a contextual term $\ell_t(i)$ that depends on the prefix $\widehat{z}_{<t}$, a position-independent semantic relevance score $r(x, c, s_i)$, and a learned task-aware membership prior σ_i that is order-agnostic.

5 Experiment

5.1 Implementation Details

Data curation. We assemble 9,872 task–skill-sequence training records over the curated skill library released with SkillBench (Li et al., 2026), organised into three groups by ordering ground. **Real anchors** are the 65 human-authored software-engineering tasks from SkillBench paired with gold skill annotations; per-task ordering is recovered from agent trajectory logs when available and from a Gemini 2.5 Pro fallback otherwise. **Single-skill synthetic** tasks (2,880 records, synthesized by Gemini 2.5 Flash) cover all 196 skills uniformly and calibrate the composer to terminate after a single skill on simple queries. **Multi-skill synthetic** tasks (6,927 records, synthesized by Gemini 2.5 Pro) cover compositions of 2–5 skills with two complementary ordering grounds: *dependency* edges, where the upstream skill's output type overlaps the downstream skill's input type, encode hard data-flow ordering, and *workflow* edges, mined from skill co-occurrence in real anchor trajectories, encode empirical ordering where no shared I/O type exists. Both edge types are sampled from a 196-node skill dependency graph. The full corpus is split 90% into train set, and 5% each for validation and test sets. Synthesis prompts, the skill dependency graph, and the deduplication and validity-check pipeline are deferred to Appendix B.

Model implementation. As introduced in §4, `SkillComposer` pairs a frozen encoder with a small task-conditioned decoder. We compare two encoder backbones: a causal LM (*Qwen3-0.6B-Base*, mean-pooled) and a retrieval-tuned dense embedding model (*Qwen3-Embedding-0.6B*, last-token pooled with the model card's instruct prefix). The decoder is a 3-layer pre-norm Transformer with hidden width 256, 4 attention heads, dropout 0.1, and cross-attention into the 196-row skill memory; the output vocabulary is the closed library plus `STOP`, `START`, and `PAD`. Two logits fusion priors are trained jointly with the sequence loss at weights $\alpha = 0.5$ and $\beta = 0.25$. Training uses AdamW (learning rate 1×10^{-4}, weight decay 0.01, batch size 64) for up to 100 epochs with patience-15 early stopping on validation Set F1. At inference, we run a width-4 beam search with length penalty 0.7 and a duplicate-skill constraint, fusing a TF-IDF retrieval prior and the set-head score into the per-step decoder logits over the library indices only. The lexical and set-fusion weights, together with a per-split stop bias that absorbs the synthetic-vs-real cardinality skew, are picked by coordinate ascent on the validation split.

5.2 Experiment Setup

Skill Composition Evaluation. We evaluate `SkillComposer` under two regimes that share data composition but differ in the held-out subset, isolating the in-domain ceiling from real-task transfer. SkillBench supplies the 196-skill library paired with 65 real tasks, and the graph-grounded synthetic corpus, all paired with deterministic verifiers built on the Harbor evaluation framework (Harbor Framework Team, 2026). The **in-distribution test** (n=494) measures the ceiling when train and test draw from the same generator. The **real-task holdout** (n=65) removes every real task from train and val, trains `SkillComposer` and baseline methods on synthetic-only data, and tests on the 65 held-out real tasks.

Baselines. We compare `SkillComposer` against three families of baselines, all predicting an ordered list of ≤ 8 skill names from the same closed library. *Retrieval baselines* include BM25, TF–IDF cosine, and Qwen3-Embedding-0.6B; for each, we report a val-tuned-k variant (best-k) and an *oracle-k* variant given the gold list length, isolating selection quality from cardinality prediction. *LLM-judge (Gemini-2.5-flash)* is a frontier-API baseline that scores all 196 skills in a single prompt populated with their names and metadata, returning the ordered shortlist directly; the model picks both *which* skills and *how many*. *SFT Qwen3-0.6B-Base* fine-tunes the full 600M backbone to generate the ordered skill sequence as text, with the 196 skill names added to the tokenizer as special tokens and greedy decoding stopping at the trained EOS class.

Metrics. We report five metrics for skill prediction quality, covering selection and ordering, and report cardinality calibration separately in Figure 4. *Set F1* is the order-agnostic F1 between predicted and gold skill sets, capturing selection quality and cardinality calibration in a single number. *Recall@5* measures gold-skill coverage in the top-5 predictions, decoupling selection from cardinality. *MRR* is the reciprocal rank of the first prediction that hits a gold skill. *nDCG@5* grades top-5 ordering by binary relevance with logarithmic discount. *Set EM* is order-agnostic exact match. For downstream task performance, we report pass rate, normalised gain, and Codex input prompt tokens (Section 5).

Task Performance Evaluation. To verify that better skill prediction translates to agent execution gains, we evaluate downstream task performance on 75 of the 88 SkillsBench tasks (Li et al., 2026), excluding 13 office and document-processing tasks dominated by the Anthropic-bundled format skills (`pdf`, `xlsx`, `pptx`, `docx`) that any retriever trivially routes by file extension and that would therefore not discriminate between skill-loading methods. We evaluate with two production-grade coding agents: **GPT-5.2-Codex** (via Azure OpenAI) and **Gemini-3-Pro-Preview** (via the Gemini CLI). Each agent runs inside the Harbor evaluation framework (Harbor Framework Team, 2026) with deterministic pytest verifiers and a 1200 s timeout; we run three attempts per task (225 trials per agent–condition) at temperature 0 and report *binary pass rate* following the SkillsBench protocol. We also report the average *input prompt tokens* per non-errored trial as a measure of context overhead. We compare `SkillComposer` to four skill conditions: *No Skills* (no procedural context), *All Skills* (the full 196-skill library injected into the prompt), *Retrieval (top-3)* (from Qwen3-Embedding), and *Retrieval (oracle)* (retrieval restricted to the gold skill set), with *Gold Skills* (the curated task-specific oracle) as the upper bound.

5.3 Skill Prediction Quality

`SkillComposer` wins in distribution at a fraction of the parameter cost. On the synthetic test (Table 1, left), `SkillComposer` leads SFT by $+2.8$ pp Set F1 and the LLM-judge by $+12.9$ pp while training $\sim$154$\times$ fewer parameters (3.9M vs 600M). The same ranking holds on MRR and nDCG@5 — the AR head produces sharper top-of-list orderings even when SFT can directly memorise gold sequences as text, while SFT retains a small edge only on Set EM. The cardinality slice (Fig. 4) further shows that `SkillComposer`'s advantage concentrates in the k=1 bucket where over-emission is most punished, yielding the highest macro-averaged Set F1. Both trained models dominate retrieval and the LLM-judge, including the oracle-k retrieval ceilings, because they predict who-and-how-many jointly while retrieval needs the gold count and the LLM-judge cannot read full skill bodies within its context budget.

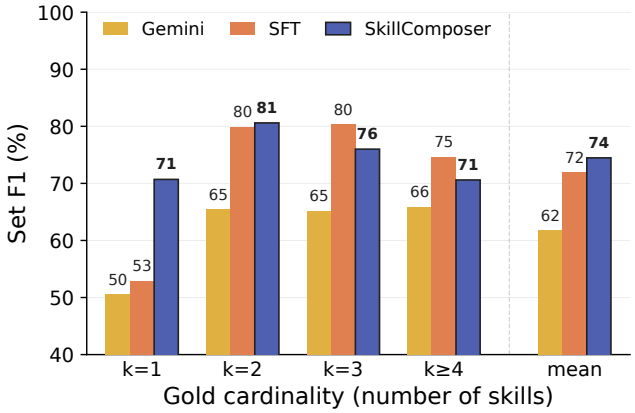


Figure 4: Cardinality robustness on the in-distribution test split. Set F1 stratified by gold cardinality k, with the macro-averaged mean on the right.

Table 1: Skill prediction quality (%). Left: in-distribution synthetic test (n=494). Right: real-task holdout (n=65); trained models are retrained on the real-task-removed partition. Best non-oracle result in **bold**; second best underlined; oracle-cardinality retrievers (in *italics*) are reported as ceilings and excluded from the ranking.

	Synthetic test					Real-task holdout				
Method	Set F1	R@5	MRR	nDCG@5	SetEM	Set F1	R@5	MRR	nDCG@5	SetEM
Retrieval (oracle-k)										
BM25	*35.3*	*35.3*	*54.4*	*38.3*	*14.8*	*55.9*	*55.6*	*73.6*	*59.2*	*33.8*
TF–IDF	*58.4*	*58.4*	*81.0*	*63.0*	*28.7*	*74.2*	*73.2*	*89.2*	*77.3*	*47.7*
Qwen3-Emb.	*49.0*	*49.0*	*69.9*	*52.0*	*20.2*	*73.4*	*72.7*	*90.9*	*76.9*	*47.7*
Retrieval (best-k)										
BM25 (k=2)	33.0	33.7	53.9	37.0	2.2	47.0	48.7	72.3	54.0	7.7
TF–IDF (k=2)	52.5	53.0	82.2	58.9	4.3	60.6	60.6	89.2	67.7	10.8
Qwen3-Emb. (k=3)	43.9	55.0	72.3	55.3	2.6	58.5	**69.2**	90.8	**73.8**	10.8
LLM-judge										
Gemini-2.5-flash	61.0	69.0	69.3	63.1	21.3	59.9	63.8	81.8	65.1	15.4
Trained models										
Qwen3-0.6B-Base	71.1	68.9	79.2	74.1	**44.9**	43.6	36.1	66.2	46.0	16.9
`SkillComposer`$_{Base}$	70.4	70.9	84.2	73.2	37.2	53.9	45.0	87.7	54.1	16.9
`SkillComposer`	**73.9**	**72.4**	**86.5**	**75.0**	41.3	**62.9**	54.7	**90.8**	63.4	**20.0**

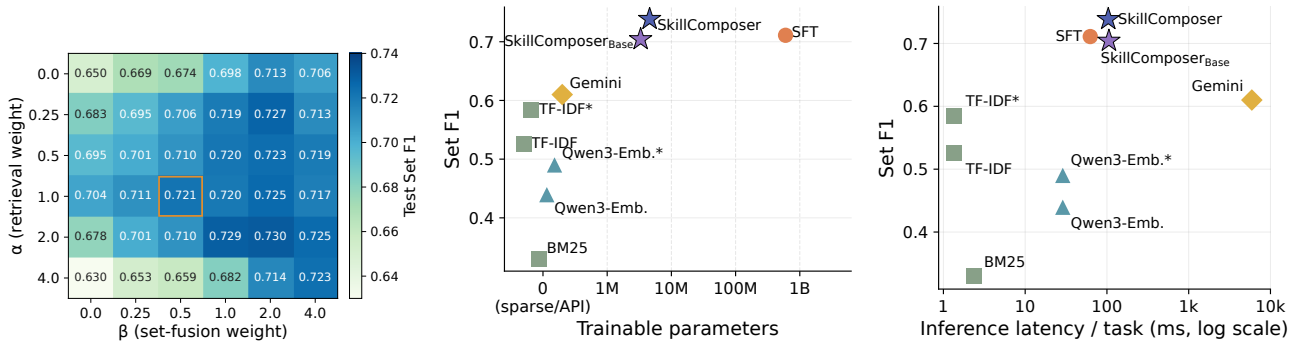
Under distribution shift, SFT degrades sharply while `SkillComposer` degrades gracefully. On the real-task holdout (Table 1, right), SFT loses 27.5 pp going from synthetic to real tasks, whereas `SkillComposer` loses only 11 pp, a $+19.3$ **pp Set F1 gap** on identical training data and library. Among predicted-k methods, only `SkillComposer` approaches the oracle-k ceilings without being told the gold count, and it remains the strongest on Set F1 even against the frontier LLM-judge. Retrieval baselines actually improve from synthetic to real tasks because real-task phrasing is closer to the skill descriptions, whereas SFT has memorised the synthetic template distribution and has no robust prior to fall back on. The frozen retrieval-tuned encoder paired with a small specialist decoder is what supplies `SkillComposer` with this transfer bias.

5.4 Downstream Task Performance

Skill prediction quality carries through to agent execution. Both agents follow the same pattern across baselines: pass rate climbs from *No Skills* to the *Gold Skills* ceiling, leaving roughly $+25$ pp of headroom for any skill-loading mechanism to close. Loading the entire library (*All Skills*) recovers only a small fraction of that headroom while inflating the Codex prompt to 1.27M input tokens, confirming that flooding the context is not enough. *Retrieval (top-3)* closes a much larger share at a smaller prompt budget, and even *Retrieval (oracle)* only matches it, showing that the remaining headroom is driven by per-task selection quality rather than retrieval recall. `SkillComposer` predicts a calibrated ordered shortlist without using oracle skill labels, reaching **45.3** / **44.0** pass rate on {Codex, Gemini}, beating both retrieval baselines and matching or exceeding *Retrieval (oracle)*, while using the smallest prompt budget (1.03M Codex tokens) among skill-loaded conditions. This closes roughly 80% of the

Table 2: Downstream task performance on SkillsBench. Pass rate follows the paper-binary protocol, *Tok.* is the average input prompt tokens per non-errored trial. Best non-oracle result in **bold**; second best underlined.

	GPT-5.2-Codex		Gemini-3-Pro	
Skill condition	Pass (%) ↑	Tok. ↓	Pass (%) ↑	Tok. ↓
Retrieval (oracle)	*44.0*	*1.13M*	*42.2*	*1.19M*
Gold Skills	*51.1*	*1.12M*	*48.4*	*1.18M*
No Skills	22.2	**0.94M**	25.8	**0.99M**
All Skills	29.3	1.27M	38.7	1.33M
Retrieval (top-3)	44.0	1.09M	41.8	1.14M
`SkillComposer`	**45.3**	1.03M	**44.0**	1.08M



(a) α, β sensitivity grid. **(b)** Trainable params vs. accuracy. **(c)** Inference latency vs. accuracy.

Figure 5: **(a)** Test Set F1 across the (α, β) decoding-weight grid; the surface is smooth and bowl-shaped around the val-selected operating point. **(b)** Compute–accuracy frontier on the synthetic test split: `SkillComposer` w/ Qwen3-Embedding is Pareto-optimal among predicted-k methods, sitting above SFT at $\sim$154$\times$ fewer trainable parameters and $\sim$25$\times$ less training compute. **(c)** Latency–accuracy frontier (1$\times$A6000, fp16, batch 1): `SkillComposer` sits in the same latency class as SFT but with higher accuracy.

headroom on both agents, confirming that upstream gains in composition prediction (Table 1) translate to real agent execution.

5.5 Ablation Study

All ablations train and evaluate on the in-distribution split; further ablations and per-layer breakdowns are reported in Appendix D.

Hyperparameter sensitivity and compute–accuracy frontier. Figure 5(a) sweeps decoding weights (α, β) on a 6×6 grid; the surface is smooth and bowl-shaped, with the val-selected operating point within 2 pp of every neighbour, indicating no fragile hand-tuning. The compute–accuracy frontier (Fig. 5b) shows `SkillComposer` matching or beating SFT while training $\sim$154$\times$ fewer parameters and using $\sim$25$\times$ less compute, and dominating retrieval and the LLM-judge by a clear margin. Inference latency (Fig. 5c) sits in the same class as SFT and is two orders of magnitude faster than the API-based judge. Together these views position `SkillComposer` as Pareto-optimal among predicted-k methods on this benchmark.

Table 3: Model component ablation.

Variant	Set F1
AR-only (no auxiliary heads)	69.3
+ set head	71.8
+ cardinality head	69.6
`SkillComposer`	**73.9**
− decode set-fusion (β=0)	65.0
− decode retrieval prior (α=0)	67.5

Each component is load-bearing. Table 3 ablates components in `SkillComposer`. Starting from the AR head alone, adding the set-membership head during training lifts Set F1 by +2.5 pp, order-agnostic gradients on every gold skill complement the AR objective. At decode time, both fusion priors are necessary: zeroing the set-fusion bias costs 7.1 pp and zeroing the lexical retrieval prior costs 4.6 pp. The auxiliary heads are reused at inference to refine the AR logits, and removing either signal degrades the predicted shortlist.

Sparse beats dense as the decode-time prior. Table 4 sweeps the retrieval prior fed into the per-step decoder logits. TF–IDF wins on Set F1 by +2.5 pp over dense Qwen3-Embedding cosine and +3.8 pp over no prior. The closed library exposes 196 short, syntactically specific skill names; token-level overlap is high-precision for distinguishing them, while dense embeddings average over broader semantic context and overgeneralise. The dense encoder is

Table 4: Decode-time retrieval prior ablation.

Decode prior	Set F1
No prior	67.5
BM25	70.0
Qwen3-Embedding	68.8
TF–IDF	**73.9**

still the right choice for the task representation: `SkillComposer` uses Qwen3-Embedding for h_x and TF–IDF as the decode-time prior, combining the strengths of both.

6 Conclusion

We formalize skill composition as task-conditioned ordered skill-sequence prediction over a closed library, jointly resolving *which* skills to load, *how many*, and in *what order*. `SkillComposer` consists of a frozen retrieval-tuned encoder and a small autoregressive decoder whose logits fuse a TF-IDF retrieval prior and a set-membership signal at inference time. With only $\sim$3.9M trainable parameters, `SkillComposer` matches SFT on an in-distribution test split and outperforms it by $+19.3$ pp Set F1 on a held-out set of real software-engineering tasks, while remaining the strongest predicted-k method against retrieval and frontier-API judges. The result suggests that, for closed agentic skill libraries, a small specialist that exploits the structure of the library is a more reliable composer than scaling up a generalist LM.

References

Anthropic. Equipping agents for the real world with agent skills. https://www.anthropic.com/engineering/equipping-agents-for-the-real-world-with-agent-skills, 2025. 1

Hao, S., Liu, T., Wang, Z., and Hu, Z. Toolkengpt: Augmenting frozen language models with massive tools via tool embeddings. *Advances in neural information processing systems*, 36:45870–45894, 2023. 3, 4

Harbor Framework Team. Harbor: A framework for evaluating and optimizing agents and models in container environments, January 2026. URL https://github.com/harbor-framework/harbor. 7, 8

Jiang, Y., Li, D., Deng, H., Ma, B., Wang, X., Wang, Q., and Yu, G. Sok: Agentic skills–beyond tool use in llm agents. *arXiv preprint arXiv:2602.20867*, 2026. 1, 3

Jiao, Z., Wang, S., Zhang, Z., Ren, X., Wang, W., Zhao, B., Wei, H., and Zhang, L. Agentic proposing: Enhancing large language model reasoning via compositional skill synthesis. *arXiv preprint arXiv:2602.03279*, 2026. 1

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. Swe-bench: Can language models resolve real-world github issues? *International Conference on Learning Representations*, 2024. 1

Li, F., Tagkopoulos, P., and Tagkopoulos, I. Skillflow: Scalable and efficient agent skill retrieval system. *arXiv preprint arXiv:2504.06188*, 2025. 1, 2, 3

Li, X., Chen, W., Liu, Y., Zheng, S., Chen, X., He, Y., Li, Y., You, B., Shen, H., Sun, J., et al. Skillsbench: Benchmarking how well agent skills work across diverse tasks. *arXiv preprint arXiv:2602.12670*, 2026. 2, 3, 4, 7, 8

Liu, D., Li, Z., Du, H., Wu, X., Gui, S., Kuang, Y., and Sun, L. Graph of skills: Dependency-aware structural retrieval for massive agent skills. *arXiv preprint arXiv:2604.05333*, 2026a. 1, 3

Liu, Y., Wang, W., Feng, R., Zhang, Y., Xu, G., Deng, G., Li, Y., and Zhang, L. Agent skills in the wild: An empirical study of security vulnerabilities at scale. *arXiv preprint arXiv:2601.10338*, 2026b. 3

Ma, Z., Huang, Z., Liu, J., Wang, M., Zhao, H., and Li, X. Automated creation of reusable and diverse toolsets for enhancing llm reasoning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 24821–24830, 2025. 3

Qian, C., Han, C., Fung, Y., Qin, Y., Liu, Z., and Ji, H. Creator: Tool creation for disentangling abstract and concrete reasoning of large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 6922–6939, 2023. 3

Rajput, S., Mehta, N., Singh, A., Hulikal Keshavan, R., Vu, T., Heldt, L., Hong, L., Tay, Y., Tran, V., Samost, J., et al. Recommender systems with generative retrieval. *Advances in Neural Information Processing Systems*, 36:10299–10315, 2023. 4

Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Hambro, E., Zettlemoyer, L., Cancedda, N., and Scialom, T. Toolformer: Language models can teach themselves to use tools. *Advances in neural information processing systems*, 36:68539–68551, 2023. 2

Shen, Y., Song, K., Tan, X., Li, D., Lu, W., and Zhuang, Y. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. *Advances in Neural Information Processing Systems*, 36:38154–38180, 2023. 3

Shen, Y., Song, K., Tan, X., Zhang, W., Ren, K., Yuan, S., Lu, W., Li, D., and Zhuang, Y. Taskbench: Benchmarking large language models for task automation. *Advances in Neural Information Processing Systems*, 37:4540–4574, 2024. 3

Su, W., Long, J., Ai, Q., Tang, Y., Wang, C., Tu, Y., and Liu, Y. Skill retrieval augmentation for agentic ai. *arXiv preprint arXiv:2604.24594*, 2026. 2, 3

Trivedi, H., Khot, T., Hartmann, M., Manku, R., Dong, V., Li, E., Gupta, S., Sabharwal, A., and Balasubramanian, N. 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*, 2024. 1

Wang, C., Yu, Z., Xie, X., Yao, W., Fang, R., Qiao, S., Cao, K., Zheng, G., Qi, X., Zhang, P., et al. Skillx: Automatically constructing skill knowledge bases for agents. *arXiv preprint arXiv:2604.04804*, 2026. 1

Wang, G., Xie, Y., Jiang, Y., Mandlekar, A., Xiao, C., Zhu, Y., Fan, L., and Anandkumar, A. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023. 1, 2, 3

Wu, X., Shen, Y., Shan, C., Song, K., Wang, S., Zhang, B., Feng, J., Cheng, H., Chen, W., Xiong, Y., et al. Can graph learning improve planning in llm-based agents? *Advances in Neural Information Processing Systems*, 37:5338–5383, 2024. 3

Xia, P., Chen, J., Wang, H., Liu, J., Zeng, K., Wang, Y., Han, S., Zhou, Y., Zhao, X., Chen, H., et al. Skillrl: Evolving agents via recursive skill-augmented reinforcement learning. *arXiv preprint arXiv:2602.08234*, 2026. 1, 2, 3

Xie, T., Zhang, D., Chen, J., Li, X., Zhao, S., Cao, R., Hua, T. J., Cheng, Z., Shin, D., Lei, F., et al. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. In *Advances in Neural Information Processing Systems*, 2024. 1

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

Yang, J., Jimenez, C. E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K., and Press, O. Swe-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, 2024. 1

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023. 2

Yuan, L., Chen, Y., Wang, X., Fung, Y. R., Peng, H., and Ji, H. Craft: Customizing llms by creating and retrieving from specialized toolsets. *arXiv preprint arXiv:2309.17428*, 2023. 3

Yue, M., Liu, Z., Yang, L., Zhang, J., Liu, Z., Chen, H., Yao, Z., Savarese, S., Xiong, C., Heinecke, S., et al. Toollibgen: Scalable automatic tool creation and aggregation for llm reasoning. *arXiv preprint arXiv:2510.07768*, 2025. 1

Zheng, Y., Zhang, Z., Ma, C., Yu, Y., Zhu, J., Wu, Y., Xu, T., Dong, B., Zhu, H., Huang, R., and Yu, G. Skillrouter: Skill routing for llm agents at scale. *arXiv preprint arXiv:2603.22455*, 2026. 2, 3

Zhou, S., Xu, F. F., Zhu, H., Zhou, X., Lo, R., Sridhar, A., Cheng, X., Ou, T., Bisk, Y., Fried, D., Alon, U., and Neubig, G. Webarena: A realistic web environment for building autonomous agents. 2024. URL https://openreview.net/forum?id=oKn9c6ytLx. 1

Zhuang, Y., Chen, X., Yu, T., Mitra, S., Bursztyn, V., Rossi, R. A., Sarkhel, S., and Zhang, C. Toolchain*: Efficient action space navigation in large language models with a* search. *arXiv preprint arXiv:2310.13227*, 2023. 3

Appendix

A Limitations and Broader Impacts

While `SkillComposer` demonstrates that structured prediction over a reusable skill library yields accurate and well-calibrated skill compositions across diverse agent task domains, our study mainly focuses on text-only task descriptions paired with a code-oriented skill library, evaluated through composition-level metrics and downstream agent benchmarks. A more comprehensive exploration of structured skill composition could incorporate multimodal task specifications (screenshots, sketches, voice instructions), interactive and long-horizon settings where the skill library is updated online, and specialized domains such as scientific workflows, robotics, and embodied agents where the composition graph extends across heterogeneous tools and physical actuators. Another direction left to future work is scaling the underlying composer beyond the small-LM and embedding backbones used here; `SkillComposer` inherits the characteristics of these components, including the linguistic priors and task coverage of the pre-training corpora, and we expect stronger backbones and larger curated skill libraries to further sharpen composition accuracy and ordering.

Regarding the broader impacts of `SkillComposer`, it advances flexible and efficient agent construction by predicting which reusable skills to compose, how many, and in what dependency order, making it well-suited for real-world scenarios where developers maintain growing skill libraries and need reliable orchestration, such as data analysis assistants, web-task automation, and database interaction. Operating at the skill abstraction (rather than raw API calls) also encourages reuse and modular auditing, reducing redundant code generation and promoting more sustainable deployment of agent systems. We follow standard practices in model development and evaluation, use only public datasets and openly released model checkpoints, and encourage responsible use in downstream applications. This work does not target any specific sensitive domain and is intended as a general-purpose framework for advancing structured skill composition for LLM agents.

B Implementation Details

This appendix expands Section 5.1 with the skill dependency graph, the Gemini prompts used for synthesis, and the deduplication and validation pipeline.

B.1 Skill Dependency Graph

The graph that grounds multi-skill synthesis has 196 nodes (one per library skill) and two structural edge types used for sampling, summarised in Table 5. *Dependency* edges connect skill pairs whose input/output schemas overlap: the upstream skill's output type matches an input type of the downstream skill, so the ordering is determined by data flow. *Workflow* edges connect skill pairs that co-occur in real-task agent trajectories; ordering follows the observed execution order, used when no shared I/O type exists.

Sampling for multi-skill synthesis draws 65 % of pairs from dependency edges and 35 % from workflow edges, matching the observed ratio of hard data-flow chains to looser plan-then-implement workflows in real anchor tasks.

Table 5: Skill dependency graph used for grounding multi-skill synthesis.

Edge type	Count
Dependency (I/O overlap)	658
Workflow (anchor co-occurrence)	266
Total	924

B.2 Synthesis Prompts

We synthesise single-skill records (Figure 6) and multi-skill records (Figure 7) with two Gemini prompts. The single-skill prompt asks Gemini 2.5 Flash to produce five tasks per call across distinct domains and difficulty levels, never naming the target skill in the task description, so the composer has to recover skill identity from semantics rather than surface form. The multi-skill prompt asks Gemini 2.5 Pro to compose all input skills into one realistic task, propose an execution order, and emit a permutation of the supplied skill IDs; whenever sampled skills are connected by a dependency edge, an explicit ordering constraint is injected so that the data-flow direction is preserved.

B.3 Deduplication and Validation

Every synthesised record is checked against all previously accepted records before being added to the pool, using three layers of similarity in escalating strictness. First, an exact-string match on the task identifier or instruction text rejects byte-identical duplicates. Second, character-trigram Jaccard similarity above 0.6 between two instructions is treated as near-duplicate phrasing and rejected. Third, sentence-embedding cosine similarity above 0.92, computed against the cached library embedding bank, is treated as a semantic duplicate. Records that survive deduplication are validated against the closed vocabulary: any response that adds, drops, or renames a skill — or whose `ordered_skills` field is not an exact permutation of the prompt input — is dropped. For multi-skill records, we additionally check that any dependency-edge ordering constraint emitted into the prompt is respected by the returned ordering. The 90/5/5 train/val/test split is applied independently within each of the three groups (real anchors, single-skill synthetic, multi-skill synthetic) using a fixed seed (42), so the group ratio is preserved across splits and the validation/test sets are not dominated by the more numerous synthetic groups.

C Extra Analysis

C.1 Downstream Case Studies

To understand where the headline pass-rate gap in Table 2 comes from, we inspect three SkillsBench tasks (GPT-5.2-Codex, three trials each, identical agent and task definition) where `SkillComposer`, *Retrieval (top-3)*, and *Gold Skills* disagree on the supplied skill set. The three cases isolate three distinct mechanisms behind the gap.

Case 1: top-3 truncation drops a key skill, and `SkillComposer` diverges from gold. On `adaptive-cruise-control`, top-3 retrieval keeps the three obvious control skills but cuts `imc-tuning-rules` — the IMC heuristic that produces PID gains satisfying the rise-time and overshoot specs in one shot — so the agent hand-tunes gains and misses the spec on 2/3 trials. The curated *Gold Skills* set is even more telling: it bundles two I/O-format skills (`csv-processing`, `yaml-config`) that add no information for the controller-tuning bottleneck, and itself only reaches 0.33. `SkillComposer` diverges from gold, drops the I/O wrappers, and adds the substantive `imc-tuning-rules` that gold

Figure 6: Single-skill synthesis prompt (Gemini 2.5 Flash). Five tasks per call ensure scenario diversity at fixed cost; difficulty is balanced 2/2/1 across easy/medium/hard.

Single-skill synthesis

```
You write realistic task descriptions for an AI coding agent.

Given this skill:
  Name: {name}
  Description: {description}
  Body (truncated): {body_truncated}

Generate EXACTLY 5 realistic task descriptions that a user would give to
    an
AI agent, where solving each task requires this exact skill. Requirements
    :
- Do NOT name the skill directly in any task description.
- Each task must use a DIFFERENT scenario/domain (e.g., finance, IoT
  sensors, bioinformatics, logistics, social media analytics).
- Each task must use a DIFFERENT input shape (e.g., single file,
    directory
  of files, streaming data, API response, database export).
- No two tasks should share the same core use case or workflow pattern.
- Each task should be concrete enough that a solution can be evaluated.
- Prefer domain-realistic framing (a user's workflow, a data scenario,
    ...).
- Difficulty distribution: exactly 2 easy, 2 medium, and 1 hard task.

Return STRICT JSON only, no markdown fences, no extra text -- a JSON
    array
of exactly 5 objects:
[
  {"task": "...", "difficulty": "easy|medium|hard",
   "reasoning": "why this skill is needed",
   "scenario_tag": "short domain label"},
  ...
]
```

Figure 7: Multi-skill synthesis prompt (Gemini 2.5 Pro). Ordering constraints from dependency edges are injected verbatim when present; otherwise, Gemini proposes an execution order with rationale.

Multi-skill synthesis with dependency constraints

```
You write realistic tasks for an AI coding agent that require composing
multiple skills.

Given these {k} skills:
{skill_list}

Generate a realistic task description that requires ALL {k} skills to
solve, composed in a specific execution order. Requirements:
- Do NOT name the skills directly in the task description.
- Specify which skill runs first, second, etc., with brief rationale.
- The task should be concrete and domain-realistic (data pipeline,
  analysis workflow, system integration, etc.).

ORDERING CONSTRAINTS (from dependency analysis):
- "<skill_A>" MUST come before "<skill_B>" (produces data that <skill_B>
  consumes)
... (one line per dependency edge among the sampled skills)
You MUST respect these ordering constraints when determining execution
order.

Return STRICT JSON only (no markdown fences, no extra text):
{
  "task": "<concrete task description>",
  "ordered_skills": [<all skill IDs, in chosen execution order>],
  "rationale": "<why this execution order>",
  "difficulty": "easy|medium|hard"
}

IMPORTANT: ordered_skills must be a permutation of the supplied IDs --
same elements, different order based on execution dependencies.
```

itself failed to include, recovering full reward. The case shows that `SkillComposer` is not regressing to the gold key but identifying useful skills, while top-k retrieval, by construction, can always be one slot short of the skill that turns a near-miss into a pass.

Example 1. `adaptive-cruise-control`

Task. Implement an Adaptive Cruise Control simulation; the verifier checks rise-time $< 10\,\mathrm{s}$, overshoot $< 5\%$, steady-state speed error $< 0.5\,\mathrm{m/s}$, distance steady-state error $< 2\,\mathrm{m}$, and minimum gap $> 5\,\mathrm{m}$.

System	Reward	Skills supplied to agent
`SkillComposer`	**1.00**	`imc-tuning-rules`, `pid-controller`, `simulation-metrics`, `vehicle-dynamics`
Retrieval (top-3)	0.33	`pid-controller`, `simulation-metrics`, `vehicle-dynamics`
Gold Skills	0.33	`csv-processing`, `pid-controller`, `simulation-metrics`, `vehicle-dynamics`, `yaml-config`

Case 2: a leaner skill set beats gold. On `exoplanet-detection-period`, `SkillComposer` converges on the minimal preprocess $\rightarrow$ Lomb-Scargle $\rightarrow$ TLS recipe and solves the task on every trial, while the gold pack adds the redundant `box-least-squares` estimator and the heavyweight `exoplanet-workflows` wrapper, which lead the agent down a longer pipeline that overfits the stellar oscillation. This confirms that `SkillComposer` is learning which skills are actually useful, and that smaller well-chosen sets can beat larger curated ones — consistent with the *All Skills* row in Table 2, where dumping the full library hurts pass rate.

Example 2. `exoplanet-detection-period`

Task. A TESS lightcurve hides an exoplanet signal under stellar-activity oscillations; recover the planet's orbital period.

System	Reward	Skills supplied to agent
`SkillComposer`	**1.00**	`light-curve-preprocessing`, `lomb-scargle-periodogram`, `transit-least-squares`
Gold Skills	0.00	`box-least-squares`, `exoplanet-workflows`, + the three above

Case 3: short-sequence bias under-emits on long-chain tasks. On `lean4-proof`, `SkillComposer` emits a single Lean-related skill (`lean4-memories`, the snippet-level memo skill) and stops before `lean4-theorem-proving`, the tactic and Mathlib reference skill that the agent needs to discharge the inductive step. Retrieval keeps the full three-skill chain (adding `python-scala-functional`) and reaches 1.00; gold has the two-skill chain and matches `SkillComposer` at 0.67 only because of an unrelated Lean kernel error on one trial. The same one-slot-short pattern recurs on `grid-dispatch-operator` and `dapt-intrusion-detection`: whenever the gold sequence has $\geq$2–3 skills, `SkillComposer`'s predicted shortlist tends to fall a slot short, reflecting the synthetic corpus's emphasis on $\leq$ 3-skill compositions during data construction. This points to the most actionable headroom — improving the construction of long-sequence training records — and is consistent with the per-cardinality breakdown 4, where SFT (with full LM-style decoding) holds a small edge on $k \geq 3$ buckets.

Example 3. `lean4-proof`

Task. Finalise a Lean 4 proof template that $S_n = \sum_{i=0}^{n} 1/2^i \leq 2$ for all $n \in \mathbb{N}$.

System	Reward	Skills supplied to agent ($\lvert \cdot \rvert$)	
`SkillComposer`	0.67	`lean4-memories`	(1)
Retrieval (top-3)	**1.00**	`lean4-memories`, `lean4-theorem-proving`, `python-scala-functional`	(3)
Gold Skills	0.67	`lean4-memories`, `lean4-theorem-proving`	(2)

D Additional Ablations

This appendix expands Section 5.5 with a per-layer view of the synthetic test split for the deterministic baselines.

D.1 Per-layer breakdown of deterministic baselines

Table 6 reports Set F1 per data layer on the synthetic test split for the deterministic (retrieval and LLM-judge) baselines. The synthetic test contains only $n{=}4$ real-anchor records; with such a small sample, any per-layer estimate has a standard error above 0.25, so the real-anchor column should be read with caution. On the synthetic-only layers, the LLM-judge wins on the multi-skill compositional layers (free-form 81.6, graph-grounded 75.0) where the prompt's longer task descriptions give it more semantic signal, while TF–IDF and Qwen3-Embedding retrieval cluster around 0.45–0.55.

Table 6: Per-layer Set F1 on the synthetic test split (%, deterministic baselines only). Trained-model rows are omitted because the per-layer canonical predictions were not saved with matching record IDs; rerunning inference on the canonical checkpoints is left to a future revision.

Method	Real anchors	Single-skill	Multi-skill	Multi-skill (graph)	All
TF–IDF ($k{=}2$)	53.3	45.8	65.9	53.7	52.5
TF–IDF (oracle-k)	*66.7*	*58.3*	*66.8*	*57.1*	*58.4*
BM25 ($k{=}2$)	29.2	31.0	35.0	33.6	33.0
Qwen3-Emb ($k{=}3$)	51.8	36.8	50.3	46.3	43.9
Qwen3-Emb (oracle-k)	*62.5*	*52.8*	*50.4*	*46.8*	*49.0*
LLM-judge	62.5	60.5	81.6	75.0	71.3