FAMILIAR OPERATIONS, UNSEEN STRUCTURES: HOW TASK COMPOSITIONS AFFECT TASK GENERALIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models are trained on data from various tasks and can generalize to unseen test tasks by reusing knowledge from training. Different training tasks demonstrate different ways to compose the knowledge pieces, and thus are central in shaping both the knowledge pieces a model acquires and the ways it learns to reuse them. Yet how different compositions in training affect generalization to unseen tasks remains elusive. We thus introduce schema-based data composition, a methodology that constructs structurally diverse tasks from a small library of reusable operations (i.e., primitives) according to predefined composition structures (i.e., schemas). A schema specifies dependencies and output assembly independently of the particular primitives and inputs; executing an instantiated schema provides its reference output. This representation allows us to study how the structural content of training data shapes generalization. Our empirical study yields three key insights. (1) **Topology shapes learnability.** Schemas of the same size can differ substantially in difficulty, and the tested primitive-sampling policies do not extend the observed direct-training frontier. (2) **Structural complementarity enables transfer.** Training on complementary forms of depth and branching enables transfer to unseen structures. It can even outperform direct training on those structures with the same prompt budget per run. (3) **Source learnability matters.** We observe stronger transfer when complementary training schemas continue to provide learning signals. Including intermediate schemas can also help the model begin learning from harder schemas. These findings identify computational structure as a distinct dimension of training-data design for generalization to new tasks.

1 INTRODUCTION

Large language models (LLMs) have demonstrated that a single model trained on broad collections of data can perform a wide range of tasks (Brown et al., 2020; Chowdhery et al., 2023; Grattafiori et al., 2024; OpenAI, 2026). Training data are central to this paradigm: they expose models to basic pieces of knowledge, from facts and rules to elementary procedures, together with examples of how to use them. Learning from these examples can help models draw on familiar knowledge to solve tasks that were not explicitly demonstrated during training. Scaling studies establish the importance of data quantity (Kaplan et al., 2020; Hoffmann et al., 2022), but the role of training data extends beyond scale. The examples a model encounters shape both the knowledge it acquires and how it learns to apply that knowledge in unfamiliar situations.

Contemporary training corpora already span many tasks and knowledge areas. Kimi K2.5 (Kimi Team, 2026) studies joint training on large collections of text and vision data, while the MAI-Thinking-1 (The Microsoft AI Team, 2026) report examines mixtures of heterogeneous sources and their interactions. Beyond these large-scale training recipes, DoReMi (Xie et al., 2023) and RegMix (Liu et al., 2025) optimize domain mixtures, Skill-It (Chen et al., 2023) uses dependencies among skills to guide data selection, and Instruct-SkillMix generates instructions that require multiple skills (Kaur et al., 2025). To study how reusable knowledge supports new tasks, we refer to basic pieces of task knowledge as primitives, instantiated here as executable operations. Composition constructs new tasks by arranging these primitives and connecting their inputs and outputs. Covering the individual primitives does not, by itself, ensure exposure to the different structures in which they can be used together.

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.

Which compositions of primitive operations should a model train on to generalize to test tasks with unseen structures?

We introduce *schema* as explicit structures for composing primitives into tasks. A schema is a reusable directed acyclic structure that specifies how operations depend on intermediate results and how their outputs are assembled, without fixing the particular operations or inputs. For example, a request for the cheapest nonstop flight and the cheapest refundable hotel can be organized into two independent search–filter–select branches whose results are combined. Assigning primitives to a schema leads to a concrete computation procedure, and sampling an input and executing the computation produces a new training example with its reference output. The same primitive vocabulary can therefore generate tasks with schemas of different depths, branching patterns, and dependencies.

Building on this representation, we study *schema combination*: training a shared model on data generated from a selected set of schemas. Compositional augmentation and graph-guided generation already construct examples from reusable fragments and dependencies (Andreas, 2020; Maekawa et al., 2026), while studies of function compositions and expression-tree patterns show that training structure affects generalization (Yuan et al., 2025; Park et al., 2025). We investigate which schemas should be trained on jointly. Holding the primitive vocabulary, model initialization, learning algorithm, and per-run training prompt budget fixed, we vary the structural content of the training data and measure generalization to unseen arrangements of familiar operations.

We summarize our contributions:

- **Schema-controlled data construction for studying composition generalization.** We separate computational structure, primitive assignment and input to compare training schemas under a common learning protocol (example in Figure 1). Direct training exhibits a topology-dependent frontier, and none of the tested primitive-sampling policies extends it (Sections 4.1–4.2; Figure 4; Table 1).
- **Complementary schemas enable generalization beyond the direct-training frontier.** For example, at the same per-run prompt budget, combining chains and parallel one-operation branches reaches 45.2% mean accuracy on nine unseen schemas, compared with 9.5% from separate direct-training runs on the targets and 3.1% from a count-matched set of chains. Source substitutions and training trajectories link larger gains to active sources spanning both depth and branching (Section 4.3; Figures 5–7).
- **Validation on real data.** On tasks constructed from eight real-world benchmarks, schema combination training extends the direct-training frontier in every family. These results also reproduce the central finding of the string experiments: training on complementary structures enables transfer to task structures absent from the training data (Section 4.4; Figures 9–10; Table 9).

2 Composition Schema

We study deterministic string-transformation tasks: given an input and a program of named operations, the model predicts the final output. Let Σ be the character alphabet, Σ^* the set of finite strings, and $x, y \in \Sigma^*$ the input and output. The primitive vocabulary $\mathcal{B} = \{p_1^*, \ldots, p_m^*\}$ contains m operations, each mapping strings to strings once any literal arguments are fixed. A schema g specifies a directed acyclic computation, including the routing of intermediate results and output assembly. An assignment a selects an operation $a(v) \in \mathcal{B}$ at each primitive node v and records its literal arguments. Executing the assigned schema $g(\cdot; a)$ gives the reference output $y = g(x; a)$.

Our experiments use schemas $G[D_1, \ldots, D_B]$ with $B \geq 1$ parallel branches sharing input x. Branch $b \in \{1, \ldots, B\}$ applies $D_b \geq 1$ operations in sequence; all branches are evaluated and their outputs concatenated in a fixed order. The positive integers B and D_b specify branch count and branch depth, respectively, and $n = \sum_{b=1}^{B} D_b$

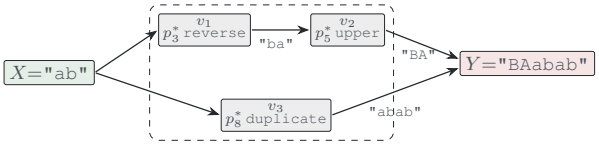

Figure 1: An example of a schema and a concrete assignment with $a(v_1) = p_3^*$, $a(v_2) = p_5^*$, and $a(v_3) = p_8^*$. By default, the outputs from previous nodes are concatenated at the output node.

counts primitive nodes, excluding input and output nodes. Figure 1 illustrates a specific task on $G[2,1]$. A schema specifies the computation required by a task, rather than a prescribed reasoning trace for the model. Separating the schema from its primitive assignment and input allows the same operations to generate tasks with different computational structures. We distinguish this construction of individual tasks from *schema combination*: a set S of schemas used to generate training examples for one shared model. Changing S changes the structural content of the training set while leaving the primitive vocabulary fixed. Section 3 describes how we instantiate these choices under a common training and evaluation protocol.

3 Experimental Setup

3.1 Data Generation

Our primitive vocabulary $\mathcal{B}$ contains 25 deterministic string operations following Yuan et al. (2025), including rotation, sorting, duplicate removal, vowel deletion, and affix insertion. Inputs are lowercase strings of length 3–10 drawn from some input distribution P_X. The name-to-operation mapping is fixed throughout the corpus: `func_j` denotes the same primitive p_j^* in every stage and run. Appendix B.1 specifies the operations, literal arguments, and primitive support.

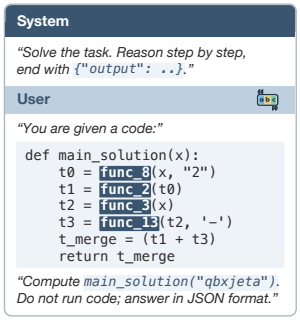

Figure 2: **A task instance.** One Stage-2 prompt for ; `func_8` and `func_13` each take an additional literal argument.

We first construct the supervised dataset $\mathcal{D}_1 = \{(q_i^{(1)}, z_i^*)\}_{i=1}^{N_1}$ from the single-primitive schema $G[1]$ in Stage 1 (SFT training). Each prompt $q_i^{(1)}$ contains an input, a primitive call, and its implementation. The target response z_i^* gives a short single-call trace followed by the answer. We generate 2,000 examples per primitive, giving $N_1 = 50{,}000$ supervised examples. Stage 2 constructs examples from a selected schema set S. In **direct training**, Stage 2 uses only examples from a single schema g, with $S = \{g\}$. A schema-combination run divides the same total prompt budget equally among the schemas in S. For each example i, let $g_i \in S$ be its selected schema. We sample a primitive assignment and an independent input, then execute the assigned computation to obtain the reference output:

$$a_i \sim \nu_{g_i}, \qquad x_i \sim P_X, \qquad y_i = g_i(x_i; a_i). \tag{1}$$

Unless otherwise stated, we use unimodal sampling, which places more weight on primitives with intermediate Stage-1 accuracy than on those with the highest or lowest accuracy (Section 4.2). For adaptive sampling, ν_{g_i} denotes the current assignment distribution conditional on earlier rollout rewards. The resulting Stage-2 dataset is $\mathcal{D}_2 = \{(q_i^{(2)}, y_i)\}_{i=1}^{N_2}$. The prompt $q_i^{(2)}$ specifies (g_i, a_i, x_i) as a program and input, but omits primitive implementations and reference intermediate values. The model must therefore use the name-to-operation mapping learned in Stage 1 to compute the final output. Figure 2 illustrates an example with schema $G[2,2]$, whose two branch outputs are concatenated in a fixed order.

3.2 Training and Evaluation

Training objectives. We use Llama-3.1-8B-Instruct (Grattafiori et al., 2024) as our base model for all experiments. Each GRPO run has a prompt budget of 49,984 sampled prompts, counted before reward-based group filtering. For policy π_θ, SFT minimizes the negative log-likelihood averaged over supervised response tokens on $\mathcal{D}_1$:

$$\mathcal{L}_{\mathrm{SFT}}(\theta) = -N_{\mathrm{tok}}^{-1} \sum_{i=1}^{N_1} \sum_{t=1}^{T_i^*} \log \pi_\theta\big(z_{i,t}^* \mid q_i^{(1)}, z_{i,<t}^*\big). \tag{2}$$

Here $T_i^* = |z_i^*|$ is the target response length in tokens and $N_{\mathrm{tok}} = \sum_{i=1}^{N_1} T_i^*$; prompt tokens are masked from the loss. Starting from θ_{SFT} obtained from SFT, each Stage-2 run uses GRPO (Shao et al., 2024) on $\mathcal{D}_2$, with the selected schema set S fixed throughout the run. Supervision comes only

from final-output correctness; reference intermediate values are not training targets. For each prompt $q_i = q_i^{(2)}$, we sample $K = 16$ responses $z_{ik} \sim \pi_{\theta_{\text{old}}}(\cdot \mid q_i)$, where θ_{old} is the policy before the update. We use the exact-match reward $R(z, y) = \mathbf{1}[\text{out}(z) = y]$, where $\text{out}(z)$ is the parsed output and y the reference string. Unparseable responses receive zero reward. For rewards $r_{ik} = R(z_{ik}, y_i)$, the group mean and standard deviation are:

$$\bar{r}_i = K^{-1} \sum_{k=1}^{K} r_{ik}, \qquad \sigma_i^2 = K^{-1} \sum_{k=1}^{K} (r_{ik} - \bar{r}_i)^2. \tag{3}$$

We discard groups with $\sigma_i = 0$. For retained prompts $i \in \mathcal{I}$, the advantages and token probability ratios are:

$$A_{ik} = \frac{r_{ik} - \bar{r}_i}{\sigma_i}, \qquad w_{ik,t}(\theta) = \frac{\pi_\theta(z_{ik,t} \mid q_i, z_{ik,<t})}{\pi_{\theta_{\text{old}}}(z_{ik,t} \mid q_i, z_{ik,<t})}. \tag{4}$$

We maximize the clipped GRPO objective:

$$\mathcal{J}_{\text{GRPO}}(\theta) = \mathbb{E}_{i,k,t}\big[\min\{w_{ik,t}(\theta)A_{ik},\ \text{clip}(w_{ik,t}(\theta), 1-\epsilon, 1+\epsilon)A_{ik}\}\big]. \tag{5}$$

The average $\mathbb{E}_{i,k,t}$ weights retained prompts equally, responses within each prompt equally, and tokens within each response equally; $T_{ik} = |z_{ik}|$ is the response length. We take one update per freshly sampled batch, holding the advantages and θ_{old} fixed during differentiation and using no KL or entropy penalty. If filtering removes every group, we skip the update.

Evaluation. We generate 1,000 fresh tasks per schema using new primitive assignments and inputs, and decode one greedy response per task. A response is correct only if its parsed output matches the reference string exactly; unparseable responses count as incorrect. More than 99.7% of responses are parseable in every final evaluation. Training-curve evaluations use 256 tasks per schema. Appendix B.2 gives the decoding settings, parsing rule, and additional pass@k evaluation protocol. A schema lies within the **direct-training frontier** if its greedy test accuracy exceeds 4% after direct training on this schema under the specified protocol and per-run prompt budget. We repeat each string-training condition with three random seeds. Unless otherwise stated, scalar accuracies are first averaged across the three seeds for each schema; reported means then average these seed-averaged accuracies over the specified evaluation schemas. Training curves show the mean across the three seeds, with bands indicating one sample standard deviation at shared checkpoints. The tool-call experiments use the protocol specified in Section 4.4 and Appendix B.4.

4 Schema Generalization

4.1 Part I: Direct Training and Transfer Across Schemas

Transfer from chains is limited on larger schemas. We start from the model that already learned primitives after Stage 1 training. We first train on a linear schema, which applies primitive operations sequentially without branching. The resulting model transfers to other schemas with the same number of operations at $n = 2$ and $n = 3$, but yields little transfer at $n = 4$ and $n = 5$ (Figure 3). At $n = 2$, accuracy peaks near 29% at about 18k sampled prompts and declines to 15.2% by the end of training; the spread across seeds contracts after 35k sampled prompts. At $n = 3$, the spread persists longer, as different seeds begin to improve at different points in training. At $n = 4$ and $n = 5$, both the mean and the spread remain low, and additional sampled prompts do not produce sustained transfer.

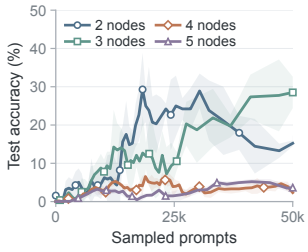

Figure 3: **Transfer from a linear schema.** Mean test accuracy over target schemas with the same node count.

Topology changes learnability even at the same task size. We next train each schema directly and evaluate it on the same structure (Figure 4, top). At $n = 2$, $G[2]$ and $G[1, 1]$ finish at similar accuracy. At $n = 3$, the fully parallel schema $G[1, 1, 1]$ finishes highest, followed by $G[3]$ and then $G[2, 1]$. At $n = 4$, every branched schema finishes above the chain $G[4]$. The ordering changes again at $n = 5$: $G[5]$, $G[4, 1]$, and $G[3, 2]$ exceed 4% accuracy, whereas all four schemas with at least three branches remain below this threshold. Node count alone therefore does not determine direct-training performance.

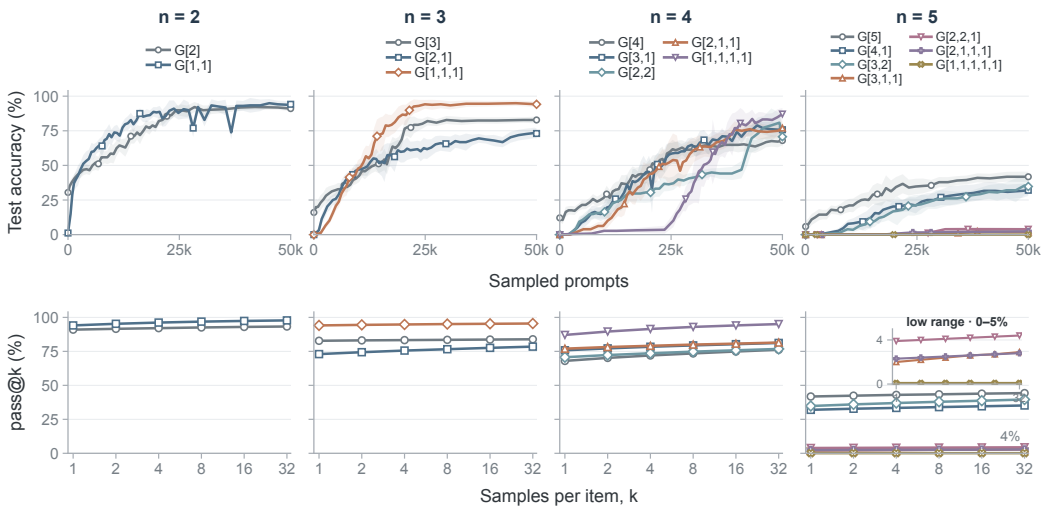

Figure 4: **Direct training on individual schemas.** Columns group schemas by node count, and each schema is trained and tested on its own shape. *Top:* test accuracy over cumulative sampled prompts. *Bottom:* the $k = 1$ point reports greedy accuracy; the remaining points report pass@k estimated from 32 stochastic responses, for $k \in \{2, 4, 8, 16, 32\}$. All schemas through $n = 4$ learn under direct training. At $n = 5$, the four schemas with at least three branches remain at most 4.4%, even at $k = 32$.

Additional samples yield little improvement on the hardest schemas. Greedy accuracy alone cannot tell whether a model cannot solve a schema or produces a correct answer only rarely. We therefore evaluate pass@k through $k = 32$ in the bottom row of Figure 4. Through $n = 4$, every schema already has high accuracy at $k = 1$. At $n = 5$, pass@k rises with k for three schemas, but the four schemas with at least three branches remain near zero; for these four, pass@32 stays close to greedy accuracy. The trainer samples groups of different rollouts for each prompt in our settings (Section 3). When pass@16 is near zero, a group is almost always uniformly incorrect. The group filter removes such groups before the update, so they don't provide valid gradient during the optimization updates. The frontier therefore marks where groups containing both correct and incorrect rollouts become too rare to train on. Increasing k at evaluation yields little improvement on these hardest schemas.

4.2 Part II: Effects of primitive sampling

We next fix the schema and vary the support and difficulty-level weights of the primitive-assignment distribution ν_g. We rank the 23 unary primitives by Stage-1 accuracy, reserve three as Stage-2 probes, and divide the remaining 20 into four difficulty quartiles L_1–L_4. Appendix B.2.2 reports the support-restriction experiment; Table 1 compares seven sampling policies on eighteen branched schemas. Within the direct-training frontier, unimodal exceeds barbell by 10.2 percentage points on average, and hard-heavy exceeds easy-heavy by 8.2 points. Adaptive sampling places most of its mass on the middle difficulty levels and approaches the unimodal weighting profile. Beyond the frontier, no policy changes accuracy by more than 1.2 percentage points, and none brings an additional schema above the 4% threshold. Primitive sampling therefore changes performance within the observed frontier but does not extend it.

4.3 Part III: Schema combination determines what becomes learnable

Direct training and primitive reweighting leave several target structures difficult to learn. We now change which schemas contribute to the training data. A schema-combination run trains one shared model on a selected set S, dividing the same total prompt budget equally among its sources.

We begin with two complementary source sets: the depth-axis set contains $G[1]$ through $G[4]$, and the branch-axis contains $G[1]$, $G[1,1]$, $G[1,1,1]$, and $G[1,1,1,1]$. Their union contains seven distinct schemas, all inside the direct-training frontier. We evaluate on the nine uniform schemas $G_{B,D} = G[D, \ldots, D]$ with $B, D \in \{2, 3, 4\}$, none of which appears in these training combinations.

Table 1: **Primitive-sampling policies affect accuracy within, but not beyond, the direct-training frontier.** Entries report accuracy changes from the same schema's uniform-sampling run, in percentage points. Bars show the weights assigned to the four difficulty quartiles. The final column averages over the ten schemas inside the frontier; † and ‡ mark the two paired policy comparisons.

		inside the frontier (n = 2–6)										beyond it (n = 5–6)								
ν_g	policy																			Mean (inside)
	Unimodal	1.1	2.3	4.7	1.6	5.7	5.9	3.5	2.9	6.9	2.2	−0.4	0.0	0.3	0.0	0.8	−0.1	0.2	0.0	3.7†
	Adaptive	1.0	2.4	4.7	3.5	4.2	3.0	2.7	0.7	2.3	0.1	0.0	−0.7	0.3	0.0	−0.3	−0.5	0.0	0.1	2.5
	Hard-heavy	−4.7	−2.5	−0.1	0.3	1.7	2.2	2.1	0.7	3.7	3.1	0.1	0.1	−0.4	0.0	−1.2	−0.1	0.4	0.0	0.7‡
	Uniform	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Easy→hard	−0.1	−0.4	−1.2	−2.7	1.0	−1.9	−1.6	−1.5	1.7	−0.5	1.0	−0.7	0.9	0.0	−1.2	−0.3	0.0	0.0	−0.7
	Barbell	−3.3	−7.5	−7.1	−7.7	−7.2	−8.3	−7.4	−7.2	−7.1	−2.8	−0.1	−0.5	0.9	−0.1	−0.6	−0.5	−0.3	0.1	−6.6†
	Easy-heavy	−2.4	−11.3	−11.1	−9.4	−8.9	−9.0	−5.1	−6.6	−7.5	−4.1	−0.5	−0.4	0.2	−0.1	−0.3	−0.4	−0.4	0.3	−7.5‡

This comparison tests whether jointly training on complementary structures enables transfer to tasks that require both depth and branching.

4.3.1 Schema combinations unlock unseen schemas

Complementary schemas enable transfer beyond the direct-training frontier. Training on the depth-axis set alone reaches 0.7% mean accuracy over the nine unseen targets, and training on the branch-axis set alone reaches 8.6%. Their union reaches 45.2% at the same total prompt budget (Figure 5, a1–a3). For comparison, separate direct-training runs on the nine targets, each receiving the full per-run budget, reach a mean of 9.5%. The seven targets with only 0.0–0.6% accuracy under direct training reach 12.6–60.9% under the combination, despite their structures being absent from its training data.

More source schemas do not, by themselves, explain the gain. To control for source count, c1 trains on seven chains, $G[1]$ through $G[7]$, using the same total budget. Its mean accuracy on the nine targets is only 3.1%. Conversely, b2 combines just four sources, $G[4]$ and the three multi-branch schemas in the branch-axis set, and reaches 22.1%, with every target above 4%. Thus, seven sources confined to one structural axis transfer less effectively than four sources spanning depth and branching.

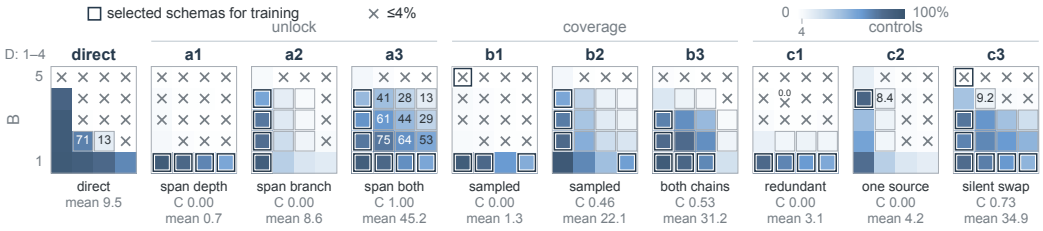

Figure 5: **Schema combinations enable generalization to unseen structures.** Columns indicate depth $D = 1, \ldots, 4$, and rows indicate branch count $B = 1, \ldots, 5$. Frames mark selected training schemas, and crosses indicate accuracy at or below 4%. All reported means use the nine targets with $B, D \in \{2, 3, 4\}$, which are absent from the training combinations. Each direct-training cell comes from a separate run on that schema. C is the active-level coverage measure defined in Appendix B.3.3. Appendix B.3.1 lists every source set, including c1's sources outside the displayed grid.

4.3.2 Generalization of new schemas follows the activation cascade

Harder source schemas become active during joint training with fixed sampling weights. We measure each source schema's activity as the fraction of its rollout groups containing both correct and incorrect responses. A source is called active once this fraction reaches 2% in each of three consecutive windows of 1,000 sampled prompts (Appendix B.3.2). Under the combined source set (Figure 6, a3), smaller sources contribute first, followed by $G[4]$ at 15k and $G[1, 1, 1, 1]$ at 19k sampled prompts. Their activation precedes the sustained rise in accuracy on the hardest unseen target, $G[4, 4, 4, 4]$ (Figure 6, a3). This progression emerges even though all selected schemas are sampled jointly throughout the run.

Activity alone does not explain held-out performance. Replacing $G[1,1,1,1]$ with the wider $G[1,1,1,1,1]$ preserves the seven-source count, but the replacement remains below the activity threshold (c3). Mean accuracy on the unseen targets falls from 45.2% to 34.9%, while accuracy on $G[2,2,2,2]$ falls from 41.2% to 9.2%. Conversely, c2 allocates the entire budget to $G[1,1,1,1]$, which supplies mixed-reward groups throughout training, yet reaches only 4.2% on the unseen targets. Together, these comparisons link stronger transfer to source sets that provide both sustained training signal and complementary coverage of depth and branching.

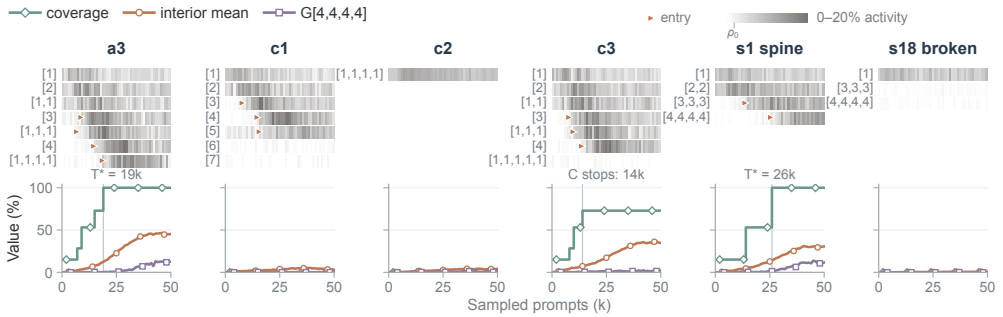

Figure 6: **Source activity and accuracy during joint training.** Lanes show each selected schema's fraction of mixed-reward groups per window of 1k sampled prompts; triangles mark activation. Curves show active level coverage, nine-cell interior mean and $G[4,4,4,4]$ accuracy; T^* marks full coverage. The last two columns are s1 and s18 of Figure 7. The s1 and s18 columns include selected interior schemas, so their interior means and $G[4,4,4,4]$ curves are not held-out evaluations.

4.3.3 Stepped combinations make larger sources trainable

Intermediate schemas can make larger selected schemas trainable. We next study stepped combinations, or spines, whose sources increase depth and branching together (Figure 7). Combinations s2 and s18 both contain three sources and share $G[3,3,3]$ and $G[4,4,4,4]$; their third source is $G[2,2]$ in s2 and $G[1]$ in s18. At the same budget and sampling shares, their mean accuracies are 35.3% and 8.3%, respectively. These means cover all twenty uniform grid cells, including selected schemas. In s1, which includes $G[2,2]$, the two large sources activate successively, whereas neither activates in s18 (Figure 6).

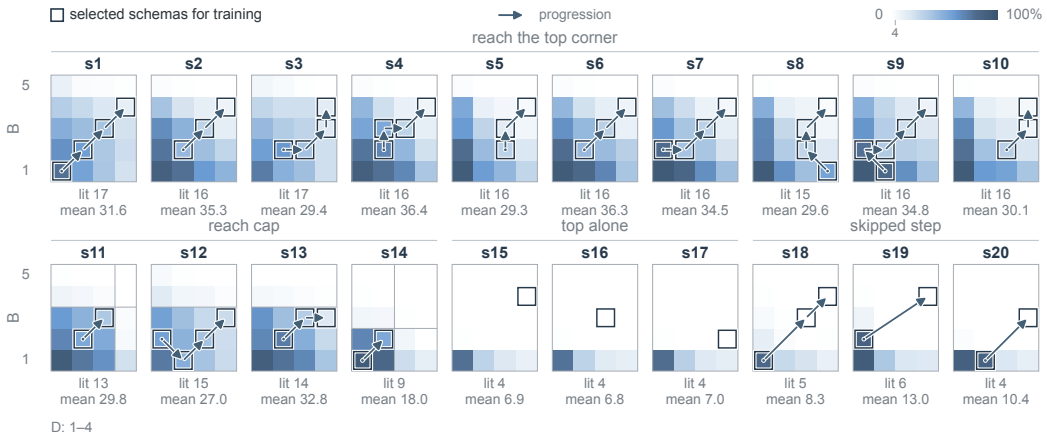

Figure 7: **Stepped source combinations produce different patterns of transfer.** Colors report accuracy on the twenty uniform evaluation schemas. Selected sources are marked at their branch count B and maximum branch depth $D = \max_b D_b$; this projection does not imply that the uniform schema at the same location was used in training. Arrows indicate structural progression among sources, all of which are sampled jointly throughout training. Counts above 4% and mean accuracies use all twenty evaluation cells, including any selected uniform schemas. Appendix B.3.4 lists the exact source sets.

Source structure shapes transfer to uniform targets. Combinations capped at two or three branches lose accuracy on wider targets (s11–s14). Allocating the entire training budget to one large schema is also insufficient: each of s15–s17 leaves only four of the twenty evaluation cells above 4%. These results distinguish the structural range reached by a collection of sources from the nominal size of its largest member. The source sets in s3–s5 contain only schemas whose branch depths differ, yet they bring 16–17 uniform evaluation schemas above the 4% threshold. All twenty uniform schemas are absent from the training data in these runs. Transfer therefore does not require a selected source to match the full branch-depth pattern of the target.

4.4 Real-world validation

We construct planning tasks from the tool catalogues of FuncBenchGen, TaskBench, API-Bank, RestBench, BFCL, TRAJECT-Bench, NESTFUL, and ToolHop (Maekawa et al., 2026; Shen et al., 2024; Li et al., 2023; Song et al., 2023; Patil et al., 2025; He et al., 2025; Basu et al., 2024; Ye et al., 2025). Each task assigns type-compatible tools to a schema and executes the resulting plan to obtain a reference. Figure 8 illustrates $G[3,3]$: two branches independently search, filter, and select a flight and a hotel. Correctness requires matching the tools, arguments, dependencies, and output bindings of the reference plan, while permitting intermediate-variable renaming and reordering of independent calls. Appendix B.4 specifies the protocol, and Appendix C gives concrete examples.

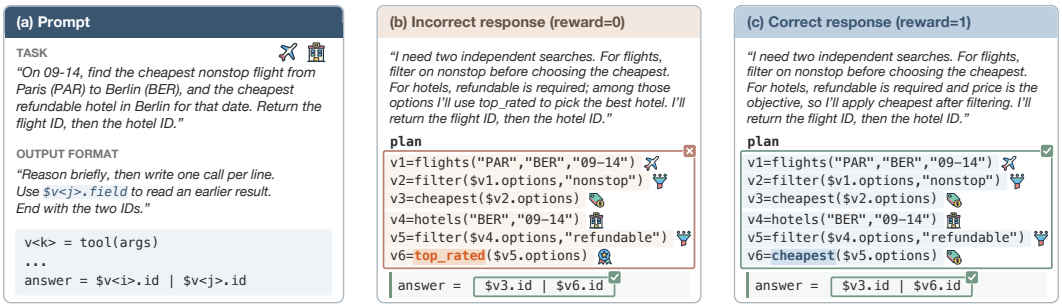

Figure 8: **A tool-composition task with two three-call branches.** (a) Request and response-format excerpt; the catalogue is omitted. (b,c) Brief reasoning followed by a complete plan. The plan in (b) chooses `top_rated` where the request requires `cheapest`; (c) selects the required tool. Green answer boxes indicate correctly formed output bindings; correctness also requires matching the calls that produce them.

Within each tool family, we compare direct training on each of the sixteen schemas with a single model jointly trained on seven schemas: four chains, $G[1]$ through $G[4]$, and three parallel schemas, $G[1,1]$, $G[1,1,1]$, and $G[1,1,1,1]$, each with one operation per branch.

Schema combination extends the direct-training frontier in every tool family. Figure 9 compares performance on unseen schemas whose direct-training accuracy is at most 4%. On this subset, mean accuracy rises from 0.18% to 15.40% in FuncBenchGen, from 0.14% to 12.86% in API-Bank, and from 0.15% to 1.95% in ToolHop. The corresponding numbers of unseen schemas brought above the threshold are six of six, seven of eight, and one of eight. The combination therefore expands structural coverage across all eight families, but the magnitude of that expansion varies substantially.

Average transfer improves across families, while the hardest structures remain uneven. Averaging the per-schema changes in Table 9 over all nine unseen targets gives positive gains in every family, ranging from 2.4 percentage points in ToolHop to 18.2 points in FuncBenchGen. The gains are substantial on several intermediate structures: $G[2,2]$ improves by 7.1–28.5 points across the eight families, and $G[2,2,2]$ improves by 8.2–32.9 points. Improvements on $G[4,4,4,4]$ are smaller, ranging from 0.0 to 6.1 points. Thus, complementary training structures support transfer across tool families without eliminating the differences in difficulty among target structures.

Figure 9 reports mean accuracies on a fixed held-out subset beyond each family's direct-training frontier. Using the same subset for both conditions makes the direct and combination means comparable within each family. Table 9 instead reports per-schema accuracy changes over the full grid, separating selected sources from held-out targets. These two views distinguish gains on

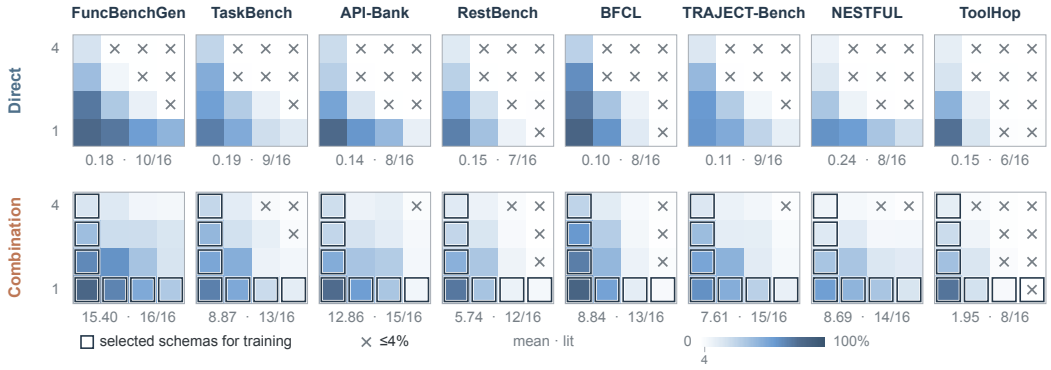

Figure 9: **Schema combinations extend the direct-training frontier across eight tool catalogues.** Direct training uses one run per schema, whereas the combination uses one run per family. Frames mark the combination's seven selected axis schemas; the nine schemas with $B, D \in \{2, 3, 4\}$ are absent from its training data. Crosses indicate accuracy at or below 4%. Reported means use the unseen schemas whose direct-training accuracy is at most 4%, while counts above 4% use all sixteen evaluation schemas, including selected axis schemas.

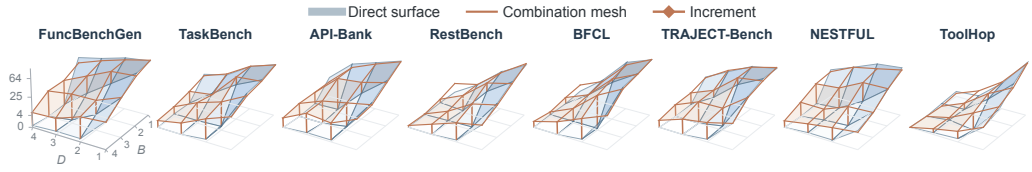

Figure 10: **Schema combinations raise accuracy beyond the direct-training frontier.** Blue surfaces show direct training; orange meshes show the combination on the same grid of branch counts and depths. Stems mark positive differences. Accuracy uses a square-root scale on the vertical axis; the near corner is $G[4, 4, 4, 4]$.

previously difficult targets from changes on the training structures under the same per-run prompt budget.

Broader transfer can come at the expense of selected training schemas. Figure 10 and Table 9 show that gains on unseen structures often accompany lower accuracy on the selected axes. For example, FuncBenchGen loses 11.8 points on $G[4]$ while gaining 27.7 points on $G[2, 2]$; NESTFUL loses 11.6 points on $G[2]$ while gaining 28.5 points on $G[2, 2]$. Unlike a direct-training run, the combination divides its budget among seven sources. Its advantage is therefore broader transfer to unseen plan structures, rather than uniformly higher accuracy on every schema. This reproduces the central pattern of the string experiments: the structural content of the training set changes which more complex tasks become learnable.

5 Conclusion

We introduced schema-based task composition to build tasks with different structures from a small set of reusable primitives. Our experiments show that learning some compositions can help a model solve others that it has not seen during training. In particular, training on chains and parallel branches together enables transfer to tasks that require both depth and branching, even outperforming direct training on those tasks with the same prompt budget per run. The choice of training schemas matters: transfer is stronger when complementary schemas continue to provide learning signals, and intermediate schemas can help the model begin learning from harder ones. Experiments on multiple real-world tasks built from public benchmark tools show similar benefits of schema combination. These findings suggest that choosing suitable training compositions can help models reuse familiar operations in new tasks.

AI USE STATEMENT

In this work, we used generative AI tools for editing assistance (grammar, spelling, and word choice). We have reviewed all AI-assisted content, and we take responsibility for the final content of this work.

REFERENCES

Jacob Andreas. Good-Enough Compositional Data Augmentation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 7556–7566, 2020. arXiv:1904.09545.

Sanjeev Arora and Anirudh Goyal. A Theory for Emergence of Complex Skills in Language Models. *arXiv preprint arXiv:2307.15936*, 2023.

Kinjal Basu, Ibrahim Abdelaziz, Kiran Kate, Mayank Agarwal, Maxwell Crouse, Yara Rizk, Kelsey Bradford, Asim Munawar, Sadhana Kumaravel, Saurabh Goyal, Xin Wang, Luis A. Lastras, and Pavan Kapanipathi. NESTFUL: A benchmark for evaluating LLMs on nested sequences of API calls. *arXiv preprint arXiv:2409.03797*, 2024.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Rafael Cabral, Tuan Manh Do, Xuejun Yu, Wai Ming Tai, Zijin Feng, and Xin Shen. ProofFlow: A Dependency Graph Approach to Faithful Proof Autoformalization. In *International Conference on Learning Representations*, 2026. arXiv:2510.15981.

Mark Chen, Jerry Tworek, Heewoo Jun, et al. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374*, 2021.

Mayee Chen, Nicholas Roberts, Kush Bhatia, Jue Wang, Ce Zhang, Frederic Sala, and Christopher Ré. Skill-it! a data-driven skills framework for understanding and training language models. *Advances in Neural Information Processing Systems*, 36:36000–36040, 2023.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *Journal of machine learning research*, 24(240):1–113, 2023.

Jui-Hui Chung, Ziyang Cai, Zihao Li, et al. Goedel-Architect: Streamlining Formal Theorem Proving with Blueprint Generation and Refinement. *arXiv preprint arXiv:2606.06468*, 2026.

Róbert Csordás, Kazuki Irie, and Jürgen Schmidhuber. The Devil Is in the Detail: Simple Tricks Improve Systematic Generalization of Transformers. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 2021. arXiv:2108.12284.

Kefan Dong, Arvind V. Mahankali, and Tengyu Ma. Formal Theorem Proving by Rewarding LLMs to Decompose Proofs Hierarchically. *arXiv preprint arXiv:2411.01829*, 2024.

Nouha Dziri, Ximing Lu, Melanie Sclar, et al. Faith and Fate: Limits of Transformers on Compositionality. In *Advances in Neural Information Processing Systems*, volume 36, 2023. arXiv:2305.18654.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Pengfei He, Zhenwei Dai, Bing He, Hui Liu, Xianfeng Tang, Hanqing Lu, Juanhui Li, Jiayuan Ding, Subhabrata Mukherjee, Suhang Wang, Yue Xing, Jiliang Tang, and Benoit Dumoulin. TRAJECT-Bench: A trajectory-aware benchmark for evaluating agentic tool use. *arXiv preprint arXiv:2510.04550*, 2025.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. An empirical analysis of compute-optimal large language model training. *Advances in neural information processing systems*, 35:30016–30030, 2022.

Dieuwke Hupkes, Verna Dankers, Mathijs Mul, and Elia Bruni. Compositionality Decomposed: How Do Neural Networks Generalise? *Journal of Artificial Intelligence Research*, 67:757–795, 2020. arXiv:1908.08351.

Albert Q. Jiang, Sean Welleck, Jin Peng Zhou, et al. Draft, Sketch, and Prove: Guiding Formal Theorem Provers with Informal Proofs. In *International Conference on Learning Representations*, 2023. arXiv:2210.12283.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

Simran Kaur, Simon Park, Anirudh Goyal, and Sanjeev Arora. Instruct-skillmix: A powerful pipeline for llm instruction tuning. In *International Conference on Learning Representations*, volume 2025, pp. 28842–28983, 2025.

Daniel Keysers, Nathanael Schärli, Nathan Scales, et al. Measuring Compositional Generalization: A Comprehensive Method on Realistic Data. In *International Conference on Learning Representations*, 2020. arXiv:1912.09713.

Najoung Kim and Tal Linzen. COGS: A Compositional Generalization Challenge Based on Semantic Interpretation. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing*, 2020. arXiv:2010.05465.

Kimi Team. Kimi K2.5: Visual Agentic Intelligence. *arXiv preprint arXiv:2602.02276*, 2026. URL https://arxiv.org/abs/2602.02276.

Lingjing Kong, Xin Liu, Guangyi Chen, Martin Q. Ma, Xiangchen Song, Yuekai Sun, Mikhail Yurochkin, Taylor W. Killian, Ruslan Salakhutdinov, Kun Zhang, Eric P. Xing, and Zhengzhong Liu. From Reasoning Traces to Reusable Modules: Understanding Compositional Generalization in Language Model Reasoning. In *International Conference on Machine Learning*, 2026. arXiv:2606.18089.

Brenden M. Lake. Compositional Generalization through Meta Sequence-to-Sequence Learning. In *Advances in Neural Information Processing Systems*, volume 32, 2019. arXiv:1906.05381.

Brenden M. Lake and Marco Baroni. Generalization without Systematicity: On the Compositional Skills of Sequence-to-Sequence Recurrent Networks. In *International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 2873–2882, 2018.

Guillaume Lample, Timothée Lacroix, Marie-Anne Lachaux, et al. HyperTree Proof Search for Neural Theorem Proving. In *Advances in Neural Information Processing Systems*, volume 35, 2022. arXiv:2205.11491.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. API-Bank: A Comprehensive Benchmark for Tool-Augmented LLMs. *arXiv preprint arXiv:2304.08244*, 2023.

Qian Liu, Xiaosen Zheng, Niklas Muennighoff, Guangtao Zeng, Longxu Dou, Tianyu Pang, Jing Jiang, and Min Lin. Regmix: Data mixture as regression for language model pre-training. In *International Conference on Learning Representations*, volume 2025, pp. 38305–38339, 2025.

Seiji Maekawa, Jackson Hassell, Pouya Pezeshkpour, Tom Mitchell, and Estevam Hruschka. Towards reliable benchmarking: A contamination free, controllable evaluation framework for multi-step LLM function calling. In *International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=UKYCJixSFt. arXiv:2509.26553.

OpenAI. Gpt-6 astra system card. https://deploymentsafety.openai.com/gpt-6-astra, September 2026. Published September 3, 2026.

Simon Park, Simran Kaur, and Sanjeev Arora. How Does RL Post-training Induce Skill Composition? A Case Study on Countdown. *arXiv preprint arXiv:2512.01775*, 2025.

Shishir G. Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. The berkeley function calling leaderboard (BFCL): From tool use to agentic evaluation of large language models. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 48371–48392. PMLR, 2025.

Stanislas Polu and Ilya Sutskever. Generative Language Modeling for Automated Theorem Proving. *arXiv preprint arXiv:2009.03393*, 2020.

Zhihong Shao, Peiyi Wang, Qihao Zhu, et al. DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models. *arXiv preprint arXiv:2402.03300*, 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. In *Advances in Neural Information Processing Systems*, volume 37, pp. 4540–4574, 2024. Datasets and Benchmarks Track.

Yifan Song, Weimin Xiong, Dawei Zhu, Wenhao Wu, Han Qian, Mingbo Song, Hailiang Huang, Cheng Li, Ke Wang, Rong Yao, Ye Tian, and Sujian Li. RestGPT: Connecting Large Language Models with Real-World RESTful APIs. *arXiv preprint arXiv:2306.06624*, 2023.

The Microsoft AI Team. MAI-Thinking-1: Building a Hill-Climbing Machine. Technical report, Microsoft AI, 2026. URL https://microsoft.ai/pdf/mai-thinking-1.pdf.

Haiming Wang, Huajian Xin, Zhengying Liu, et al. Proving Theorems Recursively. In *Advances in Neural Information Processing Systems*, volume 37, 2024a. arXiv:2405.14414.

Haiming Wang, Huajian Xin, Chuanyang Zheng, et al. LEGO-Prover: Neural Theorem Proving with Growing Libraries. In *International Conference on Learning Representations*, 2024b. arXiv:2310.00656.

Sang Michael Xie, Hieu Pham, Xuanyi Dong, Nan Du, Hanxiao Liu, Yifeng Lu, Percy S Liang, Quoc V Le, Tengyu Ma, and Adams Wei Yu. Doremi: Optimizing data mixtures speeds up language model pretraining. *Advances in Neural Information Processing Systems*, 36:69798–69818, 2023.

Kaiyu Yang, Aidan M. Swope, Alex Gu, et al. LeanDojo: Theorem Proving with Retrieval-Augmented Language Models. In *Advances in Neural Information Processing Systems, Datasets and Benchmarks Track*, volume 36, 2023. arXiv:2306.15626.

Junjie Ye, Zhengyin Du, Xuesong Yao, Weijian Lin, Yufei Xu, Zehui Chen, Zaiyuan Wang, Sining Zhu, Zhiheng Xi, Siyu Yuan, Tao Gui, Qi Zhang, Xuanjing Huang, and Jiecao Chen. ToolHop: A query-driven benchmark for evaluating large language models in multi-hop tool use. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 2995–3021. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.acl-long.150.

Dingli Yu, Simran Kaur, Arushi Gupta, Jonah Brown-Cohen, Anirudh Goyal, and Sanjeev Arora. Skill-Mix: A Flexible and Expandable Family of Evaluations for AI Models. In *International Conference on Learning Representations*, 2024. arXiv:2310.17567.

Lifan Yuan, Weize Chen, Yuchen Zhang, Ganqu Cui, Hanbin Wang, Ziming You, Ning Ding, Zhiyuan Liu, Maosong Sun, and Hao Peng. From $f(x)$ and $g(x)$ to $f(g(x))$: Llms learn new skills in rl by composing old ones. *arXiv preprint arXiv:2509.25123*, 2025.

Haoyu Zhao, Simran Kaur, Dingli Yu, Anirudh Goyal, and Sanjeev Arora. Can Models Learn Skill Composition from Examples? In *Advances in Neural Information Processing Systems*, volume 37, 2024. arXiv:2409.19808.

Appendix

CONTENTS

1 Introduction **1**

2 Composition Schema **2**

3 Experimental Setup **3**

3.1 Data Generation . . . 3

3.2 Training and Evaluation . . . 3

4 Schema generalization **4**

4.1 Part I: Direct training and transfer across schemas . . . 4

4.2 Part II: Effects of primitive sampling . . . 5

4.3 Part III: Schema combination determines what becomes learnable . . . 5

4.3.1 Schema combinations unlock unseen schemas . . . 6

4.3.2 Generalization of new schemas follows the activation cascade . . . 6

4.3.3 Stepped combinations make larger sources trainable . . . 7

4.4 Real-world validation . . . 8

5 Conclusion **9**

A Related Work **14**

B Additional Details **14**

B.1 Data generation . . . 14

B.2 Hyperparameters . . . 16

B.2.1 Additional results . . . 16

B.2.2 Primitive support and sampling policies . . . 16

B.3 Schema combinations and activity . . . 18

B.3.1 Selected schemas and evaluation sets . . . 18

B.3.2 Activity measurement . . . 19

B.3.3 Active level coverage . . . 19

B.3.4 Spine source sets . . . 19

B.4 Tool-composition tasks and evaluation . . . 19

C Case study **21**

A Related Work

Compositional generalization. Controlled benchmarks study systematic generalization by withholding novel recombinations of familiar primitives, including action sequences, meta sequence-to-sequence problems, semantic parses, and meaning representations (Lake, 2019; Keysers et al., 2020; Hupkes et al., 2020; Lake & Baroni, 2018; Kim & Linzen, 2020). Data augmentation, architectural choices, and explicit computation-graph analyses have been used to study which recombinations neural sequence models learn (Andreas, 2020; Csordás et al., 2021; Dziri et al., 2023). At language-model scale, compositionality has been investigated through emergent skill graphs, controlled skill mixtures, skill-rich training examples, expression trees, compositions of functions, and reusable reasoning modules (Yuan et al., 2025; Arora & Goyal, 2023; Yu et al., 2024; Zhao et al., 2024; Park et al., 2025; Kong et al., 2026). These settings evaluate held-out combinations, structures, or composition depths under supervised learning, in-context evaluation, and reinforcement-learning post-training.

Structured decomposition in AI for mathematics. Language-model theorem provers have used generated proof steps and guided search (Polu & Sutskever, 2020; Lample et al., 2022), formal sketches with open intermediate conjectures (Jiang et al., 2023), and premise retrieval from formal libraries (Yang et al., 2023). Later systems construct growing libraries of verified lemmas, recursively discharge intermediate conjectures, or train models to propose and prove auxiliary lemmas (Wang et al., 2024b;a; Dong et al., 2024). ProofFlow (Cabral et al., 2026) represents the logical dependencies of a natural-language proof as a directed acyclic graph and formalizes its nodes as Lean lemmas; Goedel-Architect (Chung et al., 2026) generates a dependency-graph blueprint of definitions and lemmas and assigns its open nodes to tool-equipped provers.

B Additional Details

B.1 Data generation

Primitive vocabulary The vocabulary $\mathcal{B}$ is the set of 25 deterministic string operations of Yuan et al. (2025). Table 2 describes the operations and reports their signatures and Stage 1 accuracies. Twenty-three operations take one string, with any parameters fixed inside the implementation. The two remaining operations, `interlace_str` and `recursive_interlace`, take a string and a literal argument c; c is sampled with the assignment and appears explicitly in calls such as `func_8(x, "2")` and `func_13(t2, '-')` in Figure 2. All operations map strings over printable ASCII characters to strings over the same alphabet, so intermediate outputs remain valid inputs and every primitive assignment is executable.

Stage 1 accuracy is measured on the Stage 1 model with 128 single-primitive tasks per operation, decoded greedily, with the input string drawn as in training. The two literal-taking operations are not ranked, since their difficulty depends on the literal, and they are in the training support of every run in Parts I–III.

Difficulty levels The 23 unary operations are ranked by Stage 1 accuracy in Table 2. We withhold `add_suffix`, `duplicate_every_char` and `verify_even_length` as the probe set P^*, and divide the remaining 20 into four consecutive blocks of five by rank. The probes never appear in Stage-2 training. Table 3 lists the levels and the accuracies used in the Stage 1 row of Figure 11.

Prompt templates Both stages share the system message and answer format; the user message contains a Python program defining `main_solution`. Stage 1 prompts contain a single primitive call and its implementation. In Stage 2, the program instantiates schema g with assignment a, but hides primitive implementations. The name-to-operation mapping is fixed across the corpus: `func_j` denotes the same operation in every prompt of every run. Figure 2 shows a Stage 2 instance. Node outputs are bound to `t0`, `t1`, ... in topological order, one branch after another, and the output node concatenates the branch outputs in branch order into `t_merge`. The target of a Stage 1 example is a short trace of the single call followed by the answer line, and the target string of a Stage 2 example is the value of `main_solution(x)`.

Table 2: **The primitive vocabulary.** Stage 1 accuracy is exact match on 128 single-primitive tasks, in percent. The level column gives the difficulty level of Section 4.2; P^* marks the three probes withheld during Stage 2. The two literal-taking operations are not ranked and are in the support throughout.

Operation	Description	Arguments	Stage 1 (%)	Level
`add_prefix`	prepend a fixed prefix	x	100.0	L_1
`while_rotate`	rotate left by a fixed count, one character per loop step	x	99.2	L_1
`rotate_str`	rotate left by a fixed count, by slicing	x	97.7	L_1
`reverse_words`	reverse the order of whitespace-separated words	x	96.9	L_1
`add_suffix`	append a fixed suffix	x	96.9	P^*
`alternate_case`	lower-case even positions, upper-case odd positions	x	96.9	L_1
`loop_filter_nonalpha`	drop every non-alphabetic character, in a loop	x	94.5	L_2
`vowel_to_number`	replace `a,e,i,o,u` by `1,2,3,4,5`	x	93.0	L_2
`insert_separator`	insert a fixed separator between adjacent characters	x	91.4	L_2
`remove_vowels`	delete every vowel	x	90.6	L_2
`repeat_str`	repeat the string a fixed number of times	x	89.8	L_2
`loop_concat`	concatenate the string with itself a fixed number of times, in a loop	x	85.9	L_3
`fancy_brackets`	enclose each character in brackets	x	84.4	L_3
`duplicate_every_char`	write each character twice	x	75.0	P^*
`compress_repeats`	collapse runs of equal adjacent characters to one	x	72.7	L_3
`recursive_reverse`	reverse the string by recursion	x	69.5	L_3
`mirror_str`	append the reversed string to the original	x	68.8	L_3
`shift_chars`	shift each letter by a fixed offset with wrap-around	x	59.4	L_4
`verify_even_length`	return the string if its length is even, else drop the last character	x	58.6	P^*
`sort_chars`	sort the characters	x	53.1	L_4
`deterministic_shuffle`	permute positions by a fixed multiplier permutation	x	46.1	L_4
`backchain_add_digit`	rewrite the string until it contains a digit, by a fixed-depth backtracking search	x	39.8	L_4
`backchain_palindrome`	rewrite the string into a palindrome by recursively appending its reverse, to a fixed depth	x	34.4	L_4
`interlace_str`	alternate the characters of x and c, then append the tail of the longer	x, c	–	support
`recursive_interlace`	the same interlacing computed by recursion	x, c	–	support

Table 3: **Difficulty levels.** Each level is five unary operations, listed in rank order, with the mean Stage 1 accuracy over its 640 items. The probe set P^* is scored on 384 items.

Level	Operations	Mean Stage 1 (%)
L_1	`add_prefix`, `while_rotate`, `rotate_str`, `reverse_words`, `alternate_case`	98.1
L_2	`loop_filter_nonalpha`, `vowel_to_number`, `insert_separator`, `remove_vowels`, `repeat_str`	91.9
L_3	`loop_concat`, `fancy_brackets`, `compress_repeats`, `recursive_reverse`, `mirror_str`	76.2
L_4	`shift_chars`, `sort_chars`, `deterministic_shuffle`, `backchain_add_digit`, `backchain_palindrome`	46.6
P^*	`add_suffix`, `duplicate_every_char`, `verify_even_length`	76.8

B.2 HYPERPARAMETERS

We shuffle 50,000 Stage-2 prompts and retain 49,984 in complete batches of 64, dropping the final 16. The prompt budget counts sampled prompts before reward-based filtering.

Tables 4 and 5 specify the training configurations. The third-epoch SFT checkpoint is used for the primitive evaluation in Table 2 and initializes all Stage-2 runs in Parts I–III. At each on-policy update, the sampling and trainable policies coincide, so the clipped objective has the group-normalized policy gradient at that point. The policy and support experiments vary the training assignment distribution.

Table 4: **Stage 1 (SFT) configuration.**

Base model	Llama-3.1-8B-Instruct (Grattafiori et al., 2024)
Data	2,000 single-primitive examples per operation, 50,000 in total
Epochs	3
Optimizer	AdamW, $\beta = (0.9, 0.95)$, weight decay 0.1
Learning rate	1×10^{-5}, cosine decay to 10%, 3% linear warm-up
Batch	64 sequences
Maximum sequence length	2,048 tokens
Loss	cross-entropy on target tokens only
Precision	bfloat16 parameters and activations, float32 optimizer state

Evaluation protocol We parse the last JSON object containing the single key `output` and read its value as a string. Missing or malformed objects count as incorrect. The output must match $g(x; a)$ exactly, without case or whitespace normalization. Each evaluation item uses a fresh assignment $a \sim \nu_g$, an independently drawn input x, and one greedy response. Final evaluations use 1,000 items per schema; checkpoint evaluations use 256. More than 99.7% of responses are parseable in every final evaluation.

For pass@k, we use the unbiased estimator of Chen et al. (2021) with 32 stochastic responses per item at temperature 1.0 on the 1,000 final-evaluation items. Figure 4 reports greedy accuracy at $k = 1$ and estimated pass@k for $k \in \{2, 4, 8, 16, 32\}$.

Each condition in Parts I–III (Section 4.1 4.2 and 4.3) uses three seeds, controlling prompt shuffling, assignments, inputs, rollouts, and evaluation draws. Scalar results are averaged across the three seeds. Training curves show the mean across seeds; where bands are shown, they indicate one sample standard deviation across the three seeds at shared checkpoints.

B.2.1 ADDITIONAL RESULTS

Table 6 breaks down the chain-to-branched transfer in Figure 3, alongside the chain's own accuracy and direct training on each target. At $n = 3$, the chain-trained model reaches 44.5% on $G[2, 1]$ and 12.5% on $G[1, 1, 1]$, compared with 73.0% and 94.1% under direct training. At $n = 4$ and $n = 5$, transfer is concentrated on two-branch targets; targets with three or more branches remain below 2%.

B.2.2 PRIMITIVE SUPPORT AND SAMPLING POLICIES

Support. We train four models, each sampling uniformly from one difficulty level, and a fifth on the full support. Figure 11 compares them with the Stage 1 checkpoint. Restricting support reduces accuracy on omitted levels, but the losses are asymmetric. Training on L_4 yields 88.8% and 84.8% on L_1 and L_2, compared with 98.1% and 91.9% at Stage 1. Training on L_1 instead reduces L_4 accuracy from 46.6% to 11.2%. Held-out probe accuracy is 54.7% after L_1 training and 70.3% after L_4 training, both below the Stage 1 value of 76.8%. Full-support training retains 74.5%.

Assignment policies. We fix the training budget and reweight the 20 unary training primitives across difficulty levels; the two literal-taking operations remain in the support. We compare seven policies: UNIFORM; the pair EASY-HEAVY/HARD-HEAVY; the pair BARBELL/UNIMODAL; and the schedules EASY→HARD and ADAPTIVE. The weights over $(L_1, \ldots, L_4)$ are UNIFORM $(.25, .25, .25, .25)$, EASY-HEAVY $(.40, .30, .20, .10)$ and HARD-HEAVY $(.10, .20, .30, .40)$, BARBELL $(.40, .10, .10, .40)$ and UNIMODAL $(.10, .40, .40, .10)$; the two members of each pair use the same four values and assign

Table 5: **Stage 2 (GRPO) configuration.** Each run uses one selected schema set and one assignment policy.

Initial state	Stage 1 model
Algorithm	GRPO (Shao et al., 2024), token-level clipped objective
Prompt budget	49,984 sampled prompts
Group size	16 rollouts per prompt
Group filter	discard groups with identical rewards
Batch size	64 sampled prompts (1,024 rollouts), before filtering
Updates per batch	1 (on-policy)
Clip range ε	0.2
KL / entropy coefficients	0 / 0; no reference model
Optimizer	AdamW, $\beta = (0.9, 0.999)$, weight decay 0, gradient norm clipped at 1.0
Learning rate	1×10^{-6}, constant, no warm-up
Rollout sampling	temperature 1.0, top-p 1.0
Maximum lengths	prompt: 1,024 tokens; response: 2,048 tokens; truncated responses are scored as written
Reward	1 if the parsed output equals the target string, else 0
Precision	bfloat16 parameters and activations, float32 optimizer state
Checkpoints	every 10 optimizer steps and at the end of the run
Evaluation size	256 fresh items per schema at each checkpoint; 1,000 at the end
Training seeds	$0, 1, 2$ per condition in all Stage-2 experiments

Table 6: **Transfer from chains to individual target schemas.** Last-checkpoint accuracy (%) by using 256 items per schema. Each $G[n]$-trained model is evaluated on its own schema and on branched targets with n nodes. The target mean is the endpoint in Figure 3; the last column gives direct training from Figure 4.

n	Evaluated on	Trained on $G[n]$	Own schema $G[n]$	Direct training
2		15.2	91.0	94.1
3		12.5	82.8	94.1
		44.5		73.0
	mean	28.5		
4		0.0	68.0	87.1
		0.0		77.0
		5.1		70.7
		4.3		75.8
	mean	2.3		
5		0.0	41.8	0.0
		0.4		2.3
		1.6		3.9
		0.8		2.0
		7.0		34.8
		12.1		32.0
	mean	3.6		

the larger ones to different levels. EASY→HARD moves weight from L_1 to L_4 on a fixed schedule. ADAPTIVE updates its weights every 25 optimizer steps toward levels whose success rate over the preceding 25 steps is closest to one half; ν_{g_i} then denotes the current reward-dependent assignment distribution.

The support results suggest a benefit from retaining harder primitives. Group filtering provides a different consideration: for a fixed prompt, mixed-reward groups are most frequent at intermediate rollout success probabilities (Appendix B.3.2). The advantage of UNIMODAL over BARBELL is consistent with this account, although that comparison changes weights on both the easiest and hardest levels.

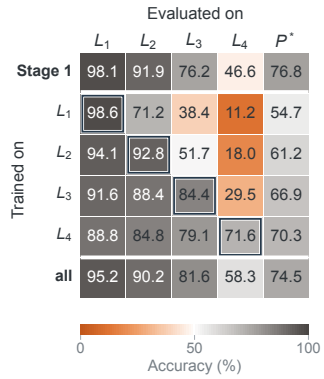

Figure 11: **Effects of restricting primitive support.** Rows compare the Stage-1 checkpoint with models trained on one difficulty quartile or on all four. Columns report accuracy on L_1–L_4 and the probe set P^*, which is withheld during Stage 2. Values are exact-match accuracy in percent.

We show each policy's change from UNIFORM on eighteen schemas in Table 1. Inside the frontier, UNIMODAL exceeds BARBELL by 10.2 percentage points on average, and HARD-HEAVY exceeds EASY-HEAVY by 8.2 points. On ◈, the seven policies span 15.8 points.

Both schedules have lower mean accuracy than UNIMODAL inside the frontier. Across the ten schemas inside the frontier, UNIMODAL improves accuracy relative to UNIFORM on all ten, while EASY-HEAVY lowers it on all ten. HARD-HEAVY changes sign with schema size, lowering accuracy on the smallest schemas and improving it on the largest. ADAPTIVE improves mean accuracy relative to UNIFORM by 2.5 percentage points, compared with 3.7 for UNIMODAL, a gap of 1.2 points. Its run-averaged weights, $(0.16, 0.32, 0.34, 0.18)$, are close to the UNIMODAL profile.

Beyond the frontier, no policy changes accuracy relative to uniform sampling by more than 1.2 percentage points. Uniform accuracy is below 4% on these eight schemas, and no policy brings an additional schema above the threshold.

B.3 SCHEMA COMBINATIONS AND ACTIVITY

B.3.1 SELECTED SCHEMAS AND EVALUATION SETS

Table 7 specifies the source sets in Figure 5. Each combination uses the initialization and assignment policy of Section 3, with its prompt budget of 49,984 sampled prompts divided evenly among its sources. The nine targets, $G[D, \ldots, D]$ with $B, D \in \{2, 3, 4\}$, are held out from every combination. The direct reference trains a separate model on each target at the full prompt budget. Accuracy means in this figure use the same nine targets. Each displayed schema accuracy is averaged over the three training seeds, and the reported means average these seed-averaged accuracies over the nine targets.

Table 7: **Source sets in Figure 5.** Each entry denotes a schema; $G[1]$ is shared by the two axes and appears only once in their union.

Panel	Combination	Selected schemas
a1	depth axis	$G[1], G[2], G[3], G[4]$
a2	branch axis	$G[1], G[1,1], G[1,1,1], G[1,1,1,1]$
a3	both axes	union of a1 and a2
b1	sampled	$G[1], G[2], G[4], G[1,1,1,1,1]$
b2	sampled	$G[4], G[1,1], G[1,1,1], G[1,1,1,1]$
b3	both chains	$G[1], G[2], G[3], G[1,1], G[1,1,1]$
c1	redundant	$G[1], G[2], G[3], G[4], G[5], G[6], G[7]$
c2	one source	$G[1,1,1,1]$
c3	silent swap	$G[1], G[2], G[3], G[4], G[1,1], G[1,1,1], G[1,1,1,1,1]$

B.3.2 ACTIVITY MEASUREMENT

We divide the sampled prompts, in training order, into windows of 1,000. Within each window, a schema's activity is the fraction of its own rollout groups containing both correct and incorrect responses. An activation event requires activity of at least $\rho_0 = 2\%$ in each of three consecutive windows. Activation is dated to the start of the first qualifying window, identified retrospectively from all three windows. The active set retains sources from their reported activation times onward. A source below the threshold can still supply occasional mixed-reward groups.

For independent rollouts from a fixed prompt with success probability p, the probability of a mixed-reward group is $1 - p^{16} - (1-p)^{16}$. Activity averages over groups from different prompts, so applying this formula to the schema's mean greedy accuracy would not recover its activity. Figure 6 reports the measured group fractions under joint sampling throughout training.

B.3.3 ACTIVE LEVEL COVERAGE

The descriptive score C summarizes depth and branch-count levels represented by active sources. Let M_D contain their maximal branch lengths and M_B their branch counts. For either level set M, write $m = \max(M \cup \{1\})$. Define $c_M(1) = 1$ and, for $t > 1$,

$$c_M(t) = \max_{1 \le r \le \min(m,t)} \left\{ \frac{r-1}{t-1} \gamma^{|\{2,\dots,r\} \setminus M|} \right\} \kappa^{\max(0,t-m)}, \qquad \gamma = 0.62, \quad \kappa = 0.28.$$

Missing intermediate levels receive a factor γ; each level beyond the largest represented level receives a factor κ. The score averages the product of the two factors over the nine interior targets:

$$C = \frac{1}{9} \sum_{B=2}^{4} \sum_{D=2}^{4} c_{M_B}(B) c_{M_D}(D).$$

Full axis coverage gives $C = 1$. In c3, the active branch counts stop at three, so the width factor at $B = 4$ is $(2/3)\kappa$ and $C = 0.729$. In b2, all nine targets exceed 4% at $C = 0.463$.

B.3.4 SPINE SOURCE SETS

Table 8 lists the sources in Figure 7. The evaluation grid has twenty uniform schemas, $B \in \{1, \dots, 5\}$ and $D \in \{1, \dots, 4\}$. Means and counts above 4% include selected uniform schemas. Source markers show branch count B and maximum branch depth $D = \max_b D_b$, not necessarily a uniform training schema. In particular, all twenty uniform schemas are held out for s3–s5. In the dynamics figure, the interior mean always uses the nine cells with $B, D \in \{2, 3, 4\}$; s1 selects three of them and s18 selects two, so those two curves are not held-out means.

B.4 TOOL-COMPOSITION TASKS AND EVALUATION

We use the eight tool catalogues listed in Section 4.4. Evaluation uses only newly generated tasks at specified schemas. Each node is assigned a tool whose required input can be bound to a compatible field of its predecessor's result. Initial inputs and remaining literal arguments are drawn from the tools' accepted values. We execute the plan in topological order and reject instances with failed calls or missing required fields. The prompt supplies the available tools, initial inputs, a natural-language request and the response format. Evaluation uses fresh tool assignments and inputs, with 1,000 tasks per shape.

A response specifies tool calls, arguments, references to earlier outputs and final output bindings. Graph matching preserves tool names, literal arguments, argument positions, referenced fields and final bindings. It permits renaming intermediate variables and reordering independent calls. Unparseable responses, missing or extra calls, and incorrect labels or bindings receive reward 0. Matching the final value alone is insufficient when the selected tool or its dependencies differ from the reference. Within each family, all compared runs start from the same model trained on single-tool tasks. Direct training assigns the full Stage 2 prompt budget to one schema; the combination divides it among the seven schemas in a3 of Table 7. The GRPO settings and prompt budget follow Section 3. Both conditions are evaluated on the sixteen uniform shapes with $B, D \in \{1, \dots, 4\}$. The nine cells with $B, D \ge 2$ are held out only for the combination.

Table 8: **The twenty spine source sets.** Full reach denotes maximum depth and branch count four. Training samples each set jointly with equal shares of the same prompt budget.

Panel	Selected schemas	Group
s1	$G[1], G[2,2], G[3,3,3], G[4,4,4,4]$	full reach
s2	$G[2,2], G[3,3,3], G[4,4,4,4]$	full reach
s3	$G[2,1], G[3,2], G[4,3,1], G[4,4,4,2]$	full reach
s4	$G[2,1], G[2,2,1], G[3,3,2], G[4,4,3,3]$	full reach
s5	$G[3,1], G[3,2,1], G[4,3,1,1]$	full reach
s6	$G[2,2], G[3,2,2], G[4,3,2,1]$	full reach
s7	$G[1,1], G[2,2], G[3,3,1], G[4,4,2,2]$	full reach
s8	$G[4], G[3,1], G[3,3,2], G[4,4,4,1]$	full reach
s9	$G[2], G[1,1], G[2,2], G[3,3,3], G[4,4,4,4]$	full reach
s10	$G[3,2], G[4,3,3], G[4,4,4,4]$	full reach
s11	$G[2,2], G[3,3,3]$	capped at 3 branches
s12	$G[1,1], G[2], G[3,3], G[4,4,4]$	capped at 3 branches
s13	$G[2,1], G[3,3,1], G[4,4,2]$	capped at 3 branches
s14	$G[1], G[2,2]$	capped at 2 branches
s15	$G[4,4,4,4]$	single large schema
s16	$G[3,3,3]$	single large schema
s17	$G[4,4]$	single large schema
s18	$G[1], G[3,3,3], G[4,4,4,4]$	skipped step
s19	$G[1,1], G[4,4,4,4]$	skipped step
s20	$G[2], G[4,4,3]$	skipped step

Figure 9 reports means over the held-out shapes whose direct-training accuracy is at most 4%, and counts above 4% over all sixteen shapes. Table 9 reports per-schema accuracy changes, with newly above-threshold held-out schemas underlined.

Table 9: **Per-schema accuracy changes for tool-call planning.** Entries are combination minus direct accuracy in percentage points. Each pair (B, D) denotes B branches of depth D. Underlines mark held-out schemas whose direct accuracy is at most 4% and whose combination accuracy exceeds 4%.

(B, D)	FuncBench Gen	TaskBench	API-Bank	RestBench	BFCL	TRAJECT-Bench	NESTFUL	ToolHop
Selected in the combination								
$(1,1)$	1.9	−0.9	−1.9	3.6	0.8	2.7	−5.6	−1.8
$(1,2)$	−5.5	0.8	−3.5	−1.6	−6.4	−1.6	−11.6	2.0
$(1,3)$	−5.7	1.7	−7.4	0.4	−1.7	2.4	0.1	2.8
$(1,4)$	−11.8	−1.9	−4.5	4.8	3.3	−0.9	−3.0	−0.5
$(2,1)$	−7.2	−3.5	−4.3	−4.2	−1.5	−5.5	0.7	−7.8
$(3,1)$	−0.1	−5.7	−3.8	−3.2	−4.5	−2.3	−0.6	4.7
$(4,1)$	−1.4	−1.0	−1.3	−0.5	−1.4	−0.2	−1.6	0.1
Held out from the combination								
$(2,2)$	27.7	16.4	19.1	15.1	7.1	14.3	28.5	7.1
$(2,3)$	27.8	−1.4	33.4	8.6	0.0	7.7	17.2	2.3
$(2,4)$	19.9	6.4	5.6	1.1	2.1	5.3	15.0	−0.2
$(3,2)$	17.4	21.4	19.9	18.5	32.9	12.8	14.7	8.2
$(3,3)$	23.5	11.7	13.9	3.9	6.1	12.9	5.3	0.6
$(3,4)$	18.8	2.6	5.0	0.1	0.5	4.0	5.9	0.0
$(4,2)$	16.2	13.8	9.4	9.5	14.2	9.7	5.3	2.7
$(4,3)$	6.8	3.6	12.1	2.9	4.4	6.4	3.0	0.8
$(4,4)$	6.1	1.3	2.5	0.1	1.0	1.4	1.2	0.0

C CASE STUDY

We illustrate task construction with six planning problems adapted from released BFCL, API-Bank and NESTFUL questions (Patil et al., 2025; Li et al., 2023; Basu et al., 2024), containing 3, 4, 5, 6, 10 and 12 tool calls. Each pairs the original question with a full rewritten input and an example plan. Blue retains source content, green copies a branch, and ochre adds or changes a step or input. Colors are not shown to the model. The final answer assembles branch outputs and adds no tool call.

The examples retain native tool names and computations. BFCL returns the displayed fields directly; API-Bank exposes successful output payloads, and NESTFUL scalars are exposed as `result`.

Problem 1. From fuel volume to duration

BFCL; $G[3]$; 3 calls

Keep the gallon conversion, then add consumption and final rounding. The one-call source becomes a three-call chain.

Original benchmark question

BFCL: multi_turn_base_88, user turn 1

"Hey, I've just filled my car up with 13.2 gallons of fuel. How much is that in liters?"

Source calls

```
gallon_to_liter(13.2)
```

Problem 1: rewritten prompt

```
SYSTEM
Solve the task using the supplied catalogue. Reason briefly, then give a complete
  tool-call plan in the required format.

USER
CATALOGUE (all tools available for this instance)
gallon_to_liter(gallon: Number) -> {liter: Number}
  Convert US gallons to liters by multiplying by 3.78541.
liter_to_gallon(liter: Number) -> {gallon: Number}
  Convert liters to US gallons by multiplying by 0.264172.
divide(a: Number, b: Number) -> {result: Number}
  Divide numerator a by nonzero denominator b.
round_number(number: Number, decimal_places: Integer = 0) -> {result: Number}
  Apply Python round(number, decimal_places).

INPUTS
recorded_fuel_gallons = 13.2
daily_consumption_liters = 5.0
decimal_places = 2

TASK
Convert the recorded 13.2 US gallons to liters. At a consumption of 5.0 liters per
  day, find how many days the fuel lasts. Round only the final duration to 2
  decimal places. Return the rounded duration.

FORMAT
Reason briefly, then write plan on its own line.
Write one call per line: v<k> = tool(args), using unique positive IDs.
Use positional arguments in signature order and double-quoted strings.
Read earlier results as $v<j>.field. Use the supplied input literals.
Do not inline arithmetic or replace intermediate results with computed literals.
Finish with answer = followed by field references separated by |, in the requested
  order.
After plan, write only calls and the answer line; include no unrelated calls.
```

Problem 1: example plan

```
The conversion supplies the fuel amount in liters. Dividing by daily use gives the
  duration, which is rounded at the end.
plan
v1 = gallon_to_liter(13.2)
v2 = divide($v1.liter, 5.0)
v3 = round_number($v2.result, 2)
answer = $v3.result
```

The example returns 9.99 days.

Problem 2. Two cities, two dependent queries

API-Bank; $G[2, 2]$; 4 calls

Supply the two tool signatures directly, eliminating catalogue search. Retain the Paris query and copy its geocoding–weather dependency for Tokyo.

Original benchmark question

API-Bank level-3 item 33 (zero-based array index)

"Check the current weather conditions in Paris."

Source calls

~~ToolSearcher~~ -> Geocoding -> ~~ToolSearcher~~ -> GetWeatherForCoordinates

Problem 2: rewritten prompt

```
SYSTEM
Solve the task using the supplied catalogue. Reason briefly, then give a complete
  tool-call plan in the required format.

USER
CATALOGUE (all tools available for this instance)
Geocoding(address: String) -> {latitude: Number, longitude: Number}
  Look up a supported place: "New York City", "San Francisco", "London", "Paris",
  or "Tokyo".
GetWeatherForCoordinates(latitude: Number, longitude: Number) -> {temperature:
  Number, humidity: Number, description: String}
  Look up the benchmark weather record at an exact supported coordinate pair.
  Temperature is in Celsius. Each call exposes its output payload; failed calls
  are rejected.

INPUTS
first_city = "Paris"
second_city = "Tokyo"

TASK
Find the weather in Paris by first resolving its coordinates. Do the same for
  Tokyo, keeping the two lookups independent. Return the temperature in Celsius
  for Paris, then Tokyo.

FORMAT
Reason briefly, then write plan on its own line.
Write one call per line: v<k> = tool(args), using unique positive IDs.
Use positional arguments in signature order and double-quoted strings.
Read earlier results as $v<j>.field. Use the supplied input literals.
Do not inline arithmetic or replace intermediate results with computed literals.
Finish with answer = followed by field references separated by |, in the requested
  order.
After plan, write only calls and the answer line; include no unrelated calls.
```

Problem 2: example plan

```
Each weather query uses both coordinates of its own city.
plan
v1 = Geocoding("Paris")
v2 = GetWeatherForCoordinates($v1.latitude, $v1.longitude)
v3 = Geocoding("Tokyo")
v4 = GetWeatherForCoordinates($v3.latitude, $v3.longitude)
answer = $v2.temperature | $v4.temperature
```

The benchmark records return 15 and 25 degrees Celsius.

Problem 3. A numeric log summary and two direct reads

BFCL; $G[3, 1, 1]$; 5 calls

Use a short supplied log in place of the dialogue history. Keep the Error search and shorten the tail; add a word-count, division and rounding chain for mean words per line.

Original benchmark question

BFCL: multi_turn_long_context_1, user turns 3–4

"Investigate within 'log.txt' for the occurrence of the keyword 'Error'."
"Finally, show the last 20 lines the file."

Source calls

```
grep("log.txt", "Error"); tail("log.txt", 20)
```

Problem 3: rewritten prompt

```
SYSTEM
Solve the task using the supplied catalogue. Reason briefly, then give a complete
  tool-call plan in the required format.

USER
CATALOGUE (all tools available for this instance)
wc(file_name: String, mode: String = "l") -> {count: Integer, type: String}
  Count lines ("l"), whitespace-separated words ("w"), or characters ("c"). The
  type field is "lines", "words", or "characters".
grep(file_name: String, pattern: String) -> {matching_lines: List[String]}
  Return lines containing the exact case-sensitive substring, in file order. The
  pattern is not a regular expression.
tail(file_name: String, lines: Integer = 10) -> {last_lines: String}
  Return the last requested number of lines, joined with newline characters.
cat(file_name: String) -> {file_content: String}
  Return the complete file content. All file names denote existing files in the
  current directory; paths are not accepted.
divide(a: Number, b: Number) -> {result: Number}
  Divide numerator a by nonzero denominator b.
round_number(number: Number, decimal_places: Integer = 0) -> {result: Number}
  Apply Python round(number, decimal_places).

INPUTS
log_file = "log.txt"
word_mode = "w"
line_count = 5
decimal_places = 2
error_marker = "Error"
tail_line_count = 2

INITIAL FILE: log.txt
INFO start
Error: sensor unavailable
Warning: retry delayed
INFO retry
```

Problem 3: rewritten prompt (continued)

```
Error: retry exhausted

TASK
Inspect log.txt in the current directory. It has 5 lines. Count its
  whitespace-separated words, divide that count by 5, and round to 2 decimal
  places to obtain mean words per line. Return that mean; the lines containing the
  exact substring "Error"; and the last 2 lines, in that order. The Error and tail
  reads are independent of the numeric branch.

FORMAT
Reason briefly, then write plan on its own line.
Write one call per line: v<k> = tool(args), using unique positive IDs.
Use positional arguments in signature order and double-quoted strings.
Read earlier results as $v<j>.field. Use the supplied input literals.
Do not inline arithmetic or replace intermediate results with computed literals.
Finish with answer = followed by field references separated by |, in the requested
  order.
After plan, write only calls and the answer line; include no unrelated calls.
```

Problem 3: example plan

```
The numeric branch counts, divides and rounds; the other two branches read the
  file directly.
plan
v1 = wc("log.txt", "w")
v2 = divide($v1.count, 5)
v3 = round_number($v2.result, 2)
v4 = grep("log.txt", "Error")
v5 = tail("log.txt", 2)
answer = $v3.result | $v4.matching_lines | $v5.last_lines
```

The 13-word log gives 2.60 words per line, followed by the Error matches and final two lines.

Problem 4. A full chain and two shorter branches

NESTFUL; $G[3, 2, 1]$; 6 calls

Change the first train's speed and crossing time. Copy the conversion prefix for a second train, and reuse the last multiplication for a third train whose speed is already in metres per second.

Original benchmark question

NESTFUL v2, sample 236711c7-5a1c-4bf5-b578-6e965b98a085

"A train running at the speed of 60 km/hr crosses a pole in 6 seconds. Find the length of the train."

Source calls

```
multiply(60, 1000) -> divide(..., 3600) -> multiply(..., 6)
```

Problem 4: rewritten prompt

```
SYSTEM
Solve the task using the supplied catalogue. Reason briefly, then give a complete
  tool-call plan in the required format.

USER
CATALOGUE (all tools available for this instance)
multiply(arg_0: Number, arg_1: Number) -> {result: Number}
  Return arg_0 multiplied by arg_1.
subtract(arg_0: Number, arg_1: Number) -> {result: Number}
  Return arg_0 minus arg_1.
add(arg_0: Number, arg_1: Number) -> {result: Number}
  Return arg_0 plus arg_1.
```

Problem 4: rewritten prompt (continued)

```
divide(arg_0: Number, arg_1: Number) -> {result: Number}
  Return arg_0 divided by nonzero arg_1.

INPUTS
train_a_speed_kmh = 90
meters_per_kilometer = 1000
seconds_per_hour = 3600
train_a_crossing_seconds = 8
train_b_speed_kmh = 72
train_c_speed_mps = 12
train_c_crossing_seconds = 5

TASK
Find the length of train A, travelling at 90 km/h and passing a pole in 8 seconds.
  Convert train B's speed of 72 km/h to metres per second. Train C travels at 12
  m/s and takes 5 seconds to pass a pole; find its length. Use 1000 metres per
  kilometre and 3600 seconds per hour. Return A's length, B's speed, then C's
  length.

FORMAT
Reason briefly, then write plan on its own line.
Write one call per line: v<k> = tool(args), using unique positive IDs.
Use positional arguments in signature order and double-quoted strings.
Read earlier results as $v<j>.field. Use the supplied input literals.
Do not inline arithmetic or replace intermediate results with computed literals.
Finish with answer = followed by field references separated by |, in the requested
  order.
After plan, write only calls and the answer line; include no unrelated calls.
```

Problem 4: example plan

```
Train A needs the full conversion-and-distance chain. Train B needs its conversion
  prefix, and train C starts from a speed already in metres per second.
plan
v1 = multiply(90, 1000)
v2 = divide($v1.result, 3600)
v3 = multiply($v2.result, 8)
v4 = multiply(72, 1000)
v5 = divide($v4.result, 3600)
v6 = multiply(12, 5)
answer = $v3.result | $v5.result | $v6.result
```

The outputs are 200 metres, 20 metres per second and 60 metres.

Problem 5. Two five-step staffing calculations

NESTFUL; $G[5, 5]$; 10 calls

Extend the worker-hour calculation with workload, efficiency and working-day adjustments, then repeat it for a second team. Each branch has five dependent calls, one level deeper than the evaluation grid.

Original benchmark question

NESTFUL v2: 06da593d-833c-457a-a042-343e08425e0c

"If 36 men can do a piece of work in 25 hours, in how mwny hours will15 men do it?"

Source calls

```
multiply(36, 25) -> divide(..., 15)
```

Problem 5: rewritten prompt

```
SYSTEM
Solve the task using the supplied catalogue. Reason briefly, then give a complete
  tool-call plan in the required format.
USER
CATALOGUE (all tools available for this instance)
multiply(arg_0: Number, arg_1: Number) -> {result: Number}
  Return arg_0 multiplied by arg_1.
subtract(arg_0: Number, arg_1: Number) -> {result: Number}
  Return arg_0 minus arg_1.
add(arg_0: Number, arg_1: Number) -> {result: Number}
  Return arg_0 plus arg_1.
divide(arg_0: Number, arg_1: Number) -> {result: Number}
  Return arg_0 divided by nonzero arg_1.
INPUTS
a_reference_workers = 36
a_reference_hours = 25
a_workload_factor = 1.2
a_workers = 15
a_efficiency = 0.8
a_hours_per_day = 8
b_reference_workers = 30
b_reference_hours = 16
b_workload_factor = 1.5
b_workers = 20
b_efficiency = 0.75
b_hours_per_day = 6
TASK
Team A's reference job needs 36 workers for 25 hours; 15 workers will do the new
  job. The new workload is 1.2 times as large, efficiency is 0.8, and the working
  day is 8 hours. Team B's reference job needs 30 workers for 16 hours. Its new
  workload factor is 1.5, with 20 workers, efficiency 0.75 and 6-hour days. For
  each team, multiply reference workers by reference hours, multiply by the
  workload factor, divide by its worker count, divide by efficiency, then divide
  by hours per day. Keep the branches independent and return A's days before B's
  days.
FORMAT
Reason briefly, then write plan on its own line.
Write one call per line: v<k> = tool(args), using unique positive IDs.
Use positional arguments in signature order and double-quoted strings.
Read earlier results as $v<j>.field. Use the supplied input literals.
Do not inline arithmetic or replace intermediate results with computed literals.
Finish with answer = followed by field references separated by |, in the requested
  order.
After plan, write only calls and the answer line; include no unrelated calls.
```

Problem 5: example plan

```
Each branch adjusts its own reference workload before converting hours to working
  days.
plan
v1 = multiply(36, 25)
v2 = multiply($v1.result, 1.2)
v3 = divide($v2.result, 15)
v4 = divide($v3.result, 0.8)
v5 = divide($v4.result, 8)
v6 = multiply(30, 16)
v7 = multiply($v6.result, 1.5)
v8 = divide($v7.result, 20)
v9 = divide($v8.result, 0.75)
v10 = divide($v9.result, 6)
answer = $v5.result | $v10.result
```

The two schedules require 11.25 and 8 working days.

Problem 6. Three complete delivered-price quotes

NESTFUL; $G[4, 4, 4]$; 12 calls

Retain percentage conversion, profit and base-price addition. Add delivery, then copy the four-step chain for two more products. The twelve-call plan must preserve all three price branches and their output order.

Original benchmark question

NESTFUL v2: bb1b666b-9820-472a-87ab-7c03411de17b

"A person buys an article at $380. At what price should he sell the article so as to make a profit of 25%?"

Source calls

```
divide(25, 100) -> multiply(380, ...) -> add(380, ...)
```

Problem 6: rewritten prompt

```
SYSTEM
Solve the task using the supplied catalogue. Reason briefly, then give a complete
  tool-call plan in the required format.

USER
CATALOGUE (all tools available for this instance)
multiply(arg_0: Number, arg_1: Number) -> {result: Number}
  Return arg_0 multiplied by arg_1.
subtract(arg_0: Number, arg_1: Number) -> {result: Number}
  Return arg_0 minus arg_1.
add(arg_0: Number, arg_1: Number) -> {result: Number}
  Return arg_0 plus arg_1.
divide(arg_0: Number, arg_1: Number) -> {result: Number}
  Return arg_0 divided by nonzero arg_1.

INPUTS
a_cost = 380
a_markup_percent = 25
a_delivery = 20
b_cost = 240
b_markup_percent = 20
b_delivery = 15
c_cost = 560
c_markup_percent = 15
c_delivery = 25
percent_denominator = 100
```

Problem 6: rewritten prompt (continued)

```
TASK
Product A costs 380 and requires profit equal to 25 percent of cost. Its delivery
  charge is 20. Product B costs 240, with markup 20 percent and delivery 15.
  Product C costs 560, with markup 15 percent and delivery 25. For each product,
  divide markup by 100, multiply cost by that fraction, add cost to the profit,
  then add delivery to the subtotal. Keep the three calculations independent.
  Return the delivered totals for A, B and C, in that order.
FORMAT
Reason briefly, then write plan on its own line.
Write one call per line: v<k> = tool(args), using unique positive IDs.
Use positional arguments in signature order and double-quoted strings.
Read earlier results as $v<j>.field. Use the supplied input literals.
Do not inline arithmetic or replace intermediate results with computed literals.
Finish with answer = followed by field references separated by |, in the requested
  order.
After plan, write only calls and the answer line; include no unrelated calls.
```

Problem 6: example plan

```
Each product converts its own markup, adds its profit to cost, and then adds
  delivery.
plan
v1 = divide(25, 100)
v2 = multiply(380, $v1.result)
v3 = add(380, $v2.result)
v4 = add($v3.result, 20)
v5 = divide(20, 100)
v6 = multiply(240, $v5.result)
v7 = add(240, $v6.result)
v8 = add($v7.result, 15)
v9 = divide(15, 100)
v10 = multiply(560, $v9.result)
v11 = add(560, $v10.result)
v12 = add($v11.result, 25)
answer = $v4.result | $v8.result | $v12.result
```

The delivered totals are 495, 303 and 669 currency units.