TOOLLIBGEN: Scalable Automatic Tool Creation and Aggregation for LLM Reasoning

Anonymous ACL submission

Abstract

Large Language Models (LLMs) can improve complex reasoning by using external tools, and recent work has explored automatically creating such tools from training data. However, these methods often produce large collections of fragmented, question-specific tools. As the toolset grows, retrieval becomes increasingly difficult due to an expanding search space and ambiguity among functionally overlapping tools. This shifts the bottleneck of automatic tool learning from tool creation to tool organization. We propose TOOLLIBGEN, a framework that automatically transforms a fragmented toolset into a structured Python library. Unlike prior methods that perform local refactoring or pairwise merging, TOOLLIBGEN conducts aggregation at a large scale. It groups functionally related tools, uses a Code Agent to synthesize higher-level abstractions, and introduces a solver-in-the-loop Reviewing Agent to verify whether the aggregated library preserves the source-task utility of the atomic tools it replaces. Experiments show that TOOLLIBGEN improves retrieval accuracy and downstream task performance over fragmented toolsets. The gains are especially strong in source-task tool reuse evaluations and remain consistent on held-out domain-near questions. We further extend TOOLLIBGEN to natural-language skill aggregation, showing that aggregation with behavioral verification can organize reusable reasoning experience beyond executable code.

1 Introduction

Large Language Models (LLMs) have demonstrated remarkable proficiency in solving complex reasoning tasks (Lewkowycz et al., 2022; Wei et al., 2022; Kojima et al., 2022). To enable LLMs to continuously learn and evolve from successful past reasoning experiences, conventional paradigms predominantly rely on Supervised Fine-Tuning (SFT) or Reinforcement Learning (RL) (Guo et al., 2025; Zhang et al., 2026). However, extensive post-training can incur catastrophic forgetting of pre-existing capabilities (Li et al., 2024), compromise established safety alignments (Qi et al., 2023), and demand prohibitively high computational costs, and cannot be directly applied to the closed-sourced LLMs. Therefore, a promising alternative paradigm has emerged: extracting reusable modules from past successful reasoning trajectories (e.g., Chain-of-Thought processes) and encapsulating them as external tools, typically in the form of executable Python functions (Yuan et al., 2023; Stengel-Eskin et al., 2024; Ma et al., 2025). It allows the LLM agent to learn from past experience without updating parameters. When confronted with novel queries, the LLM agent can simply retrieve and execute these tools.

However, the paradigm of automated tool-making approaches exposes a fatal bottleneck in scaling up. In traditional training, incorporating more data generally yields improved model performance. Paradoxically, in prevailing tool-making methodologies, we find the opposite trend, that more instances of tool-making cannot enhance performance. Because these methods tend to generate ad-hoc, question-specific tools for every individual's past experience, scaling up the experiential data accumulates an intractable mass of highly fragmented tools. This provokes two severe retrieval challenges. First, the growth of the toolset creates an overwhelmingly large search space. Second, functionally related tools with overlapping capabilities but disparate implementations cause severe retrieval ambiguity. For instance, tools synthesized to solve quadratic equations and trigonometric functions might both be arbitrarily named *"get_root"*, yet their applicable contexts diverge completely. Consequently, as more tools are extracted from past data, the retrieval process becomes increasingly error-prone, leading to performance stagnation or even degradation. Therefore, the critical bottleneck

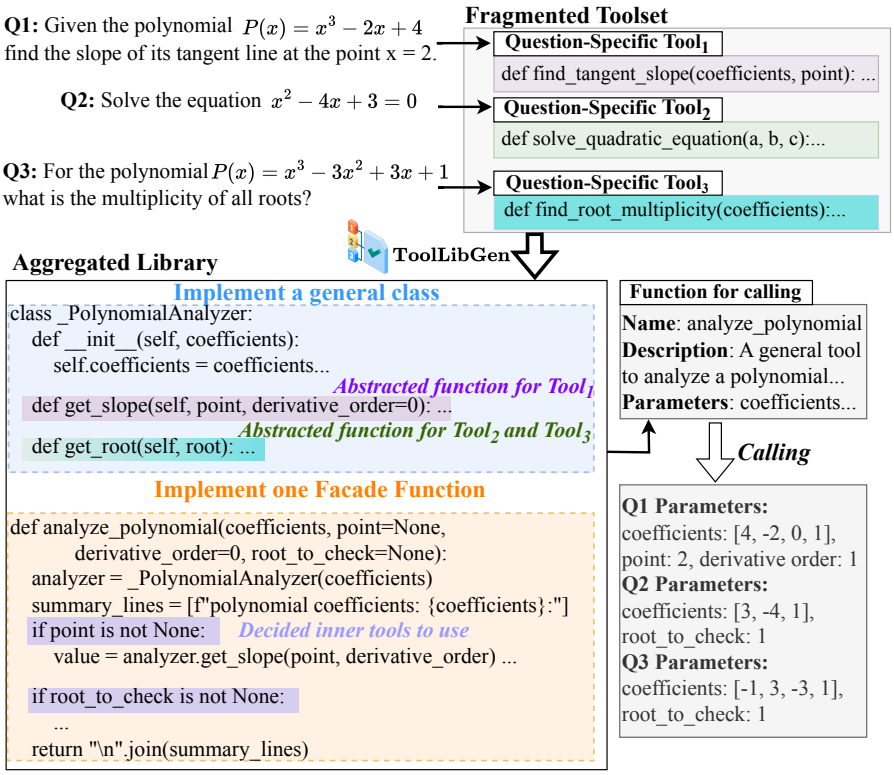

Figure 1: We obtained function-related tools $Tool_{1-3}$ from different problems and $Tool_2$ and $Tool_3$ have overlapping functionality. Through the TOOLLIBGEN, we integrated the three discrete, question-specific tools into a single class that covers all functionalities and a facade function that covers all possible parameter inputs.

in automated tool-making has decisively shifted from *"how to create tools"* to *"how to effectively organize them"* In previous research, Didolkar et al. (2024) labeled related tools but did not refactor them, while Ma et al. (2025) merged tools through pairwise comparisons, where each merging step was limited to examining only a small number of tools (5k). These approaches perform only local adjustments and thus have a limited impact. As shown in Figure 1, we argue that as the number of tools scales up(100k), a global design should be applied to refactor a large group of functionally related tools into a Python library, comprising standardized classes and public functions, through a higher level of abstraction. It can more significantly reduce the number of tools but keep the same functionality because its goal is not merely to eliminate direct, superficial redundancies, but to abstract and unify the common *core functionality* behind different tools through in-depth analysis.

In this paper, we propose a pipeline, TOOLLIBGEN, to automatically refactor a fragmented collection of ad-hoc tools into a structured library (Figure 2). The first challenge is identifying functionally related tools within a large, fragmented collection. To address tool identification, we employ a hierarchical clustering module that groups tools into coherent sub-domains. The second challenge is aggregating these related tools without losing the specific functional nuances of each implementation, as we observe that a naive, single-pass refactoring by a single LLM frequently overlooks critical details from the original tools. Therefore, the cornerstone of our approach is a multi-agent collaborative framework designed for robust, iterative refinement. TOOLLIBGEN establishes a feedback loop between a *Coding Agent*, which synthesizes a unified implementation for each tool cluster, and a *Reviewing Agent*, which rigorously validates this code for complete functional fidelity against the original tools, and generates targeted feedback for revision. This iterative process continues until the *Reviewing Agent* determines that the aggregated library is sufficiently effective for the solver LLM to answer the tool-making question and fully preserve the capabilities of the initial toolset.

We conducted extensive experiments across diverse reasoning tasks, including science, math,

and medical QA, upon multiple foundation models, including GPT-4.1 (OpenAI, 2025a), Qwen3-8B (Yang et al., 2025), and GPT-oss-20B (OpenAI, 2025c), to validate our pipeline. Our structured tool library significantly outperforms baseline methods that use fragmented ad-hoc tool collections. Our approach improves the success rate of seen tasks by an average of over 5%-10%, underscoring the superior retrieval accuracy of the library structure. Moreover, our method demonstrates strong generalization, improving accuracy on entirely unseen questions. Furthermore, we provide a detailed analysis showing that the accuracy improvement of our method stems from enhanced retrieval and conduct an experiment showing that our aggregation method can extend to natural language skill aggregation rather than only for the Python code.

2 Related Work

Tool-Augmented LLM Reasoning LLMs have evolved from relying solely on natural language reasoning to tool-augmented reasoning. LLMs are empowered to invoke external modules, such as code interpreters, search engines, and APIs (Mialon et al., 2023; Xu et al., 2023; Qu et al., 2025). For example, SciAgent (Ma et al., 2024) has demonstrated that LLMs with domain-specific toolsets can improve an LLM's performance on scientific reasoning tasks. Besides, directly having an LLM use code can improve its reasoning performance and calibration ability (Chen et al., 2022; Gao et al., 2023; Yue et al., 2023), while other works have shown that LLMs can learn to use Python code at the appropriate time to solve mathematical problems (Yue et al., 2024; Li et al., 2025). However, the widespread adoption of tool-augmented reasoning is constrained not just by the availability of domain-specific tools, but by the absence of a scalable methodology for their management.

Tool Creation and Curation for LLM Reasoning To address the scarcity of domain-specific tools, recent research has explored automatic tool creation (Zhang et al., 2024; Zheng et al., 2025; Wang et al., 2025). Seminal works such as CREATOR (Qian et al., 2023) and CRAFT (Yuan et al., 2023) established this paradigm by abstracting reusable Python functions from problem-solving traces. Follow-up work has further explored tool refactoring and curation, but remains limited in two aspects. First, methods such as CREATOR, CRAFT, TROVE, and ReGAL (Qian et al., 2023; Yuan et al., 2023; Wang et al., 2024; Stengel-Eskin et al., 2024) mainly perform local abstraction within a single question, trace, or program, and therefore do not organize a large collection of independently generated tools into a coherent library. KTCE (Ma et al., 2025) moves toward cross-tool curation through mutation, crossover, and pairwise merging, but each operation only considers a small local subset of tools and lacks a global view. As a result, existing methods have been validated only on toolsets with fewer than 5k tools, where local adjustment is still feasible. Second, prior curation methods do not explicitly optimize aggregation with a solver-in-the-loop behavioral verification criterion: tool merging is mainly guided by textual similarity, structural similarity, or local transformation rules, which do not directly test whether the refactored tool remains useful where the original tool was invoked. In contrast, TOOLLIBGEN performs cluster-level library generation at the 100k+ scale and introduces a reviewing agent that verifies whether the aggregated library preserves source-task utility; uncovered functionalities are retained as singleton tools rather than discarded. Our ablations further show that this multi-round verification and refinement process is crucial for effective aggregation.

3 TOOLLIBGEN: Scalable Tool Library Generation

3.1 Overview

We establish a systematic methodology for constructing a Python library from a QA dataset. Given a QA dataset $\mathcal{D}$, consisting of N pairs of question Q and its corresponding CoT reasoning trace, denoted as $\mathcal{D} = \{(Q_i, CoT_i)\}_{i=1}^{N}$, our system ultimately generates a Python library, denoted as $\mathcal{L}$, that consolidates all reusable deterministic logic extracted from the CoT traces.

As illustrated in Figure 2, the pipeline unfolds in three core stages. First, in the **Question-Specific Tool Creation** stage, question-specific Python functions are abstracted as tools from each question-CoT pair. After processing all questions in $\mathcal{D}$, we obtain a collection of tools $\mathcal{T} = \{t_1, \ldots, t_M\}$, where M is the number of all question-specific tools. Second, in the **Tool Clustering** stage, the workflow groups $\mathcal{T}$ into multiple clusters $\mathcal{C} = \{C_1, \ldots, C_K\}$, where each cluster consists of multiple tools, denoted as $C_k = \{t_{k,1}, \ldots, t_{k,m_k}\}$, with $m_k \in [1, K]$ and $m_k \ll M$

for all clusters. Third, in the **Tool Aggregation** stage, the tools within each cluster C_k are consolidated into cohesive Python *classes* and auxiliary *functions*, denoted as $C_k^* = \{t_{k,1}^*, \ldots, t_{k,m_k^*}^*\}$. Each aggregated tool t^* is realized as one or more Python *classes* with an associated interface *function*, encapsulating the functionalities of multiple original tools. This significantly reduces the number of tools in C_k and eliminates redundant functionality. Finally, the complete Python library is given by $\mathcal{L} = \{C_k^*\}_{k=1}^K$. Our framework employs two LLMs. One general LLM orchestrates the whole pipeline, while another LLM LLM_{solver} is used in the validation process to use the generated tools to solve questions.

3.2 Question-Specific Tool Creation

The initial phase focuses on creating question-specific tools for each question. We follow other works on the prompt template (Ma et al., 2025). The LLM first abstracts a reusable toolset from each pair of (Q_i, CoT_i). Each tool t in the toolset is a Python function, including the function signature and implementation. In practice, the LLM can occasionally generate tools that do not meaningfully contribute to reasoning. To mitigate this, we introduce a validation-and-refinement mechanism. Specifically, for each tool t, LLM_{solver} attempts to solve the original question Q_i using the tool. If tool t is used correctly and leads to a correct answer, t is eligible to be accepted into the collected toolset $\mathcal{T}$. Otherwise, the LLM receives the full context of the failure trajectory and generates an improved version of the tool, which replaces the previous version in the candidate set. By repeating this process for all questions, the creation module produces multiple question-specific tools, forming the fragmented collection of tools $\mathcal{T}$.

3.3 Tool Clustering

Given the large number of tools in the fragmented set $\mathcal{T}$, it is crucial to determine which tools should be aggregated together before aggregation. We group functionally related tools into a cluster and expect that tools within the same cluster are closely related, for example, all about the kinetics in physics. Directly clustering tools into sub-domains is challenging because the full set of domains is unknown. To address this, we first randomly select a small subset of seed tools $\mathcal{T}_{seed}$ and prompt an LLM to construct a tree-structured clustering, which encourages the LLM to produce fine-grained clusters, with nodes closer to the root corresponding to broad categories (e.g., physics) and deeper nodes representing increasingly specific subdomains (e.g., kinematics). In practice, we find that the functionalities of tools within the leaf clusters are sufficiently distinct, so it is sufficient to retain only the leaf nodes for subsequent aggregation. The initial tree produces a set of K leaf nodes $\{label_k\}_{k=1}^K$. The remaining tools are then processed iteratively for updating the current tree by creating a new leaf node or merging multiple leaf nodes. After the updating, each tool is assigned to one or more leaf nodes, resulting in the final set of clusters $\mathcal{C} = \{C_1, \ldots, C_K\}$, where each C_k contains m_k tools $(t_{k,1}, \ldots, t_{k,m_k})$.

3.4 Tool Aggregation

Given a cluster $C_k = \{t_{k,j}\}_{j=1}^{m_k}$, the goal of tool aggregation is to consolidate its question-specific tools into a smaller set of aggregated tools $C_k^* = \{t_{k,j}^*\}_{j=1}^{m_k^*}$, where $m_k^* \ll m_k$. Each aggregated tool is implemented as one or more Python *classes* with an associated interface *function*, so that multiple related atomic tools can be accessed through a unified entry point. This design turns fragmented, question-specific functions into higher-level library components, reducing redundancy while preserving the task-level utilities of the original tools.

The aggregation is performed by a Code Agent. For each cluster, the agent first enumerates the names, descriptions, and implementations of the original tools and produces a global blueprint for the cluster. The blueprint specifies the major scenarios covered by the cluster and maps individual atomic tools to more general abstractions. For example, several polynomial-related tools, such as *find_tangent_slope*, can be mapped to a shared *PolynomialAnalyzer* class with a facade function, as shown in Figure 1. Based on this blueprint, the Code Agent then generates the concrete Python implementation of the aggregated tools.

A key challenge is that aggregation cannot be judged only by textual or structural similarity between the original and refactored tools. A refactored tool is useful only if it can replace the atomic tools at the positions where they were originally invoked. We therefore introduce a solver-in-the-loop verification criterion: an aggregated library is accepted only when it preserves the source-task utility of the atomic tools it replaces. Concretely, a Reviewing Agent provides the aggregated library to

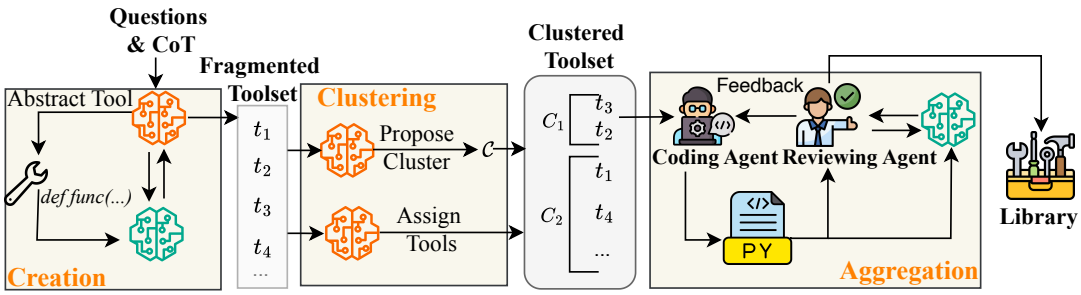

Figure 2: The pipeline of our proposed method (🧠: General LLM; 🧠: LLM_{solver}). The LLM first generates and validates question-specific tools. Then it proposes clusters and assigns each tool to specific clusters. For each cluster, a coding agent abstracts the functionality of tools and writes code, while a reviewing agent validates the code with LLM_{solver} to preserve the original functionalities.

LLM_{solver} and asks it to solve the source questions associated with the original tools. The resulting trajectories are analyzed to determine whether the aggregated library induces equivalent task behavior, i.e., whether the solver can still reach the correct answer using the aggregated abstraction instead of the original atomic tool.

If the verification fails, the Reviewing Agent produces a structured report containing the failure cases, the rationale, and concrete revision suggestions, which are passed back to the Code Agent for another round of refinement. This separation is important because the Code Agent must process the full cluster context to design the abstraction, while the Reviewing Agent only needs to test whether the generated library preserves source-task behavior; merging these roles would further enlarge the context and may degrade generation quality (Liu et al., 2025). The refinement loop continues until the aggregated library passes verification or the maximum number of iterations is reached. In the latter case, functionalities not reliably covered by the aggregated implementation are retained as singleton tools, preventing validated atomic tools from being discarded. After applying this process to all clusters in $\mathcal{C}$, we obtain the final structured Python library $\mathcal{L} = \{C_1^*, C_2^*, \ldots, C_K^*\}$ for downstream tool-augmented reasoning. All prompts are provided in Appendix G.

3.5 Tool-augmented LLM Reasoning

At inference time, we use a standard embedding-based retrieval strategy to provide relevant tools from the generated library $\mathcal{L}$. Each tool is represented by its name and description, and the LLM retrieves the top-k candidate tools using a natural-language search query. Importantly, tool use is not mandatory. The LLM is first encouraged to reason about the problem by itself and decide whether external tool support is needed. If the retrieved tools appear useful, the LLM may select one, instantiate its parameters, execute it, and incorporate the result into the final answer. Otherwise, it can ignore the retrieved tools and answer based on its own reasoning. We explicitly prompt the LLM to treat retrieved tools as optional aids rather than authoritative solutions, since retrieval may return irrelevant or only partially useful tools.

4 Experiments

4.1 Experimental Setup

Datasets and Models Our experiments are conducted across multiple domains. We constructed three libraries for different professional domains: a Science Library (from Llama-Nemotron-Post-Training-Dataset (Bercovich et al., 2025)), a Mathematics Library (from DeepMath-103K (He et al., 2025)), and a Medical QA Library (from Reason-Med (Sun et al., 2025)). More details are in Appendix B. To generate the tool library, we employed GPT-5 (OpenAI, 2025b) as the general LLM and GPT-4.1 (OpenAI, 2025a) as LLM_{solver} in our pipeline. We note that the same libraries are used to evaluate other LLMs as well.

Evaluation Setting We have two different evaluation settings: (1) **Seen Case Performance:** This setting focuses on whether a well-structured library makes it easier to retrieve useful tools for problems used in tool making. Specifically, we use the same dataset $\mathcal{D}$ for both tool making and testing. This setting is a diagnostic evaluation for tool organization.

Since each source question has at least one tool derived from its reasoning trace, this setting isolates whether a library representation enables the solver to recover and reuse the relevant prior procedure from a large tool index. (2) **Unseen Case Performance:** The second setup aims to measure the generalization ability. We use the SuperGPQA (Du et al., 2025) as the held-out set. In our evaluation, we test the specific domain with the specific library, e.g., only use the math library for math question answering. Although this dataset shares similar concepts to the library-making datasets we use (which thus shows the potential for it to be better addressed if our tool library is properly formed), the format of the problems and the difficulty level differ from those of the library-making datasets. Therefore, we consider this dataset to be out of distribution. For both settings, the Accuracy of a model is calculated based on exact match for the multiple-choice questions and simple-evals [1] for numerical answers. More details are in the Appendix B.

Baselines We compare our approach with the following baselines: (1) **CoT**: Standard prompting to elicit reasoning steps in natural language. (2) **Program-of-thought (PoT)** We equip the LLM with a Python interpreter as the only tool (Chen et al., 2022; Gao et al., 2023). (3) **Fragmented Toolset (FT)**: After generating all question-specific tools, we do not cluster or aggregate them; instead, we directly retrieve from the fragmented toolset at inference time. (4) **Clustered Toolset (CT)**: After generating all tools, we cluster them but do not aggregate them. At inference time, the LLM first predicts the relevant cluster and then retrieves the appropriate tools within that cluster. (5) **KTCE**: A prior work to iteratively optimize the toolset with function operation, such as mutation and crossover (Ma et al., 2025). We perform experiments using a range of LLMs, including GPT-4.1 (OpenAI, 2025a), GPT-oss-20B (OpenAI, 2025c), and Qwen3-8B (Yang et al., 2025). For tool retrieval, we use SFR-Embedding (Meng et al., 2024) and retrieve the top-5 tools every time.

4.2 Main Result

Table 1 presents the results, which suggest that:

Providing useful tools helps question-answering We find that augmenting LLM reasoning with tools improves question-answering performance only when the tools encode genuinely useful knowledge or methods. Compared with standard CoT, incorporating such specialized tools leads to better performance, even on unseen tasks. However, simply providing a code interpreter is not sufficient, as PoT does not always outperform CoT. This indicates that the key factor is not the availability of code execution capabilities, but the actual utility of the knowledge embedded in the tools for solving the underlying reasoning task.

Our method outperforms baselines in seen cases. In seen cases, where the problems have been previously encountered, the tool library is more likely to contain a relevant and effective tool for each problem. The primary challenge for an LLM is to find and utilize it. Our method excels over all other baselines by 5%-10% improvement. This significant gain highlights that our approach helps the effective retrieval process in finding useful tools, consistently making it easier for the LLM to access the right knowledge and tools to solve problems.

Our method even demonstrates improved performance on unseen cases. Compared with seen cases, unseen cases are much harder because the model has no prior experience with these specific problems, and there is no guarantee that a suitable tool exists in the library to solve them. Incorporating a fragmented toolset does not consistently enhance performance on unseen tasks (e.g., on the Math task with Qwen3-8B), as irrelevant tool information will hinder the problem-solving process. In contrast, our method consistently improves the performance 2-3% across different domains, which shows that our method offers suitable tools that supply effective information for this scenario.

4.3 Analysis

Why is our aggregated library better than the fragmented toolset? We first analyze why our aggregated library shows better performance. We attribute this to the retrieval issues with the fragmented toolset. In case of retrieving from the fragmented toolset, for each question Q_i, we consider the fragmented tools created from the question as the ground truths. Similarly, for retrieving from TOOLLIBGEN-produced tool library, we consider the aggregated counterparts of the extracted fragmented tools as the ground truths. A retrieval is considered successful when the LLM is able to retrieve at least one tool from the ground truths (although in practice, most questions correspond to exactly one tool). The questions are randomly se-

[1] https://github.com/openai/simple-evals

Table 1: Accuracy (%) of each method across different reasoning tasks.

Method	Retriever	Seen Case				Unseen Case in SuperGPQA			
		Nemotron	DeepMath	ReasonMed	Average	Science	Math	Medicine	Average
Solver LLM: GPT-4.1									
CoT	✗	55.8	49.3	66.7	55.9	48.6	67.0	55.8	57.1
PoT	✗	52.4	48.5	59.8	52.6	44.6	63.2	48.8	52.2
FT	✓	64.7	52.8	71.4	64.0	48.8	68.2	54.6	57.2
CT	✓	68.8	54.3	73.4	65.5	49.4	69.3	57.4	58.7
KTCE	✓	67.9	57.6	72.9	66.1	49.1	69.5	56.9	58.5
TOOLLIBGEN	✓	**75.9**	**59.3**	**75.6**	**70.3**	**51.3**	**71.4**	**59.1**	**60.6**
Solver LLM: GPT-oss-20B									
CoT	✗	52.7	40.7	46.5	46.6	49.1	72.8	40.2	54.0
PoT	✗	47.5	41.4	45.6	44.8	47.3	69.1	38.5	51.6
FT	✓	60.6	48.7	55.6	55.0	51.3	73.1	47.6	57.3
CT	✓	62.6	49.6	55.1	55.8	51.7	74.1	45.6	57.1
KTCE	✓	68.4	48.8	57.3	58.2	52.5	74.0	44.8	57.1
TOOLLIBGEN	✓	**74.3**	**52.7**	**59.4**	**62.1**	**53.5**	**76.1**	**51.8**	**60.5**
Solver LLM: Qwen3-8B									
CoT	✗	41.5	38.1	52.8	45.5	46.5	69.6	42.4	52.8
PoT	✗	40.6	39.5	46.6	43.1	44.3	66.2	37.5	49.3
FT	✓	58.8	44.3	57.4	50.9	47.8	68.9	45.9	54.2
CT	✓	59.7	44.8	57.3	51.1	48.3	69.8	45.2	54.4
KTCE	✓	63.5	46.5	58.8	52.7	48.5	70.4	45.4	54.8
TOOLLIBGEN	✓	**71.9**	**50.9**	**61.5**	**61.4**	**49.3**	**71.1**	**48.5**	**56.3**

lected from DeepMath-103k, and we use GPT-4.1 to generate the search query.

As shown on the left of Figure 3, the retrieval accuracy for a fragmented tool declines sharply as the number of tools increases. This demonstrates that an unorganized tool set fails to scale due to retrieval limitations. In contrast, our aggregated library still maintains a high retrieval rate. A case study is shown on the right of Figure 3. For a question about an *unfair die*, the retrieved tool from the fragmented toolset is related to *fair die*, but does not contribute meaningfully to solving the *unfair die* problem. By contrast, the retrieved tool from our library is a general tool for binomial Bayes questions and provides all the information required for the *unfair die* scenario. Therefore, the advantages of our library stem from merging superficially related yet functionally overlapping tools, thereby minimizing the likelihood of retrieving relevant but unhelpful tools.

Ablation Study We conduct an ablation study to validate the design of each module in TOOLLIBGEN. For both question-specific tool creation and aggregation, we confirmed the necessity of our iterative validation by comparing our performance against a single-pass alternative. For tool clustering, we demonstrated the superiority of our LLM-based method over a standard K-means baseline through manual evaluation of cluster coherence. Details are in the Appendix C. As shown in Figure 4, these results highlight the crucial role of multi-turn revision and the effectiveness of leveraging an LLM for high-quality tool clustering.

When does the library fail to help the reasoning? We conducted an error analysis of GPT-4.1's errors in tool-augmented reasoning based on TOOLLIBGEN. From DeepMath-103k, we sampled 50 erroneous tool-augmented cases for analysis. The error types can be divided into: (1) *Retrieving Error*: the error was due to failing to retrieve useful tools; (2) *Selection Error*: retrieval succeeded but the LLM failed to select the appropriate tool among the retrieved ones; (3) *Parameters Error*: the LLM supplied incorrect parameters when using the tool; (4) *Analysis Error*: the LLM over-relied on its own beliefs during reasoning and did not leverage the tool's information. The error types are shown in the Figure 4. The most significant challenge is the *Analysis Error*. This arises because many problems are complex, and our tools may not be able to provide a direct answer. They can offer effective analyses to help the LLM in reasoning. However, the LLM often fails to leverage these analyses and instead produces an incorrect result based on its own belief. This demonstrates that although tools can assist the LLM by providing analysis, there is still a critical need to enhance the LLM's inherent capabilities to handle complex problems.

Generalization to Natural Language Skill Aggregation We further examine whether TOOLLIBGEN generalizes beyond executable Python tools. We extract 268k natural-language atomic skills,

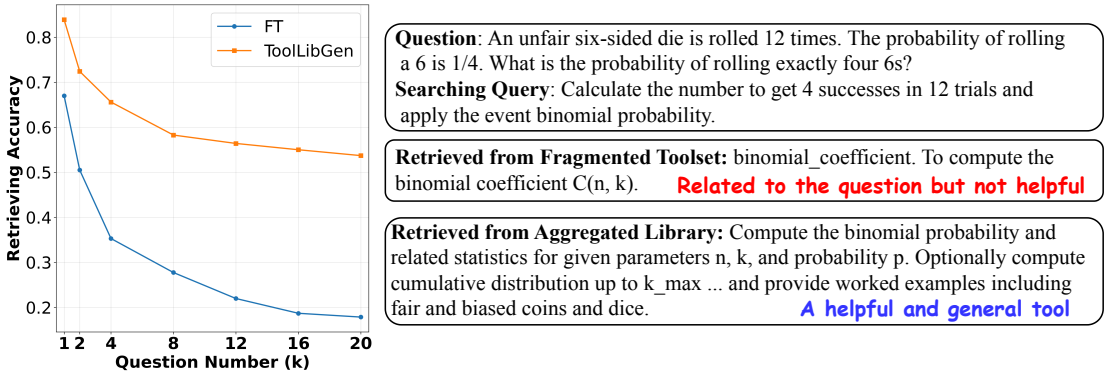

Figure 3: (Left) The retriever accuracy changes as the number of questions for tool-making increases from 1k to 20k. (Right) A case study showing how TOOLLIBGEN facilitates tool retrieval.

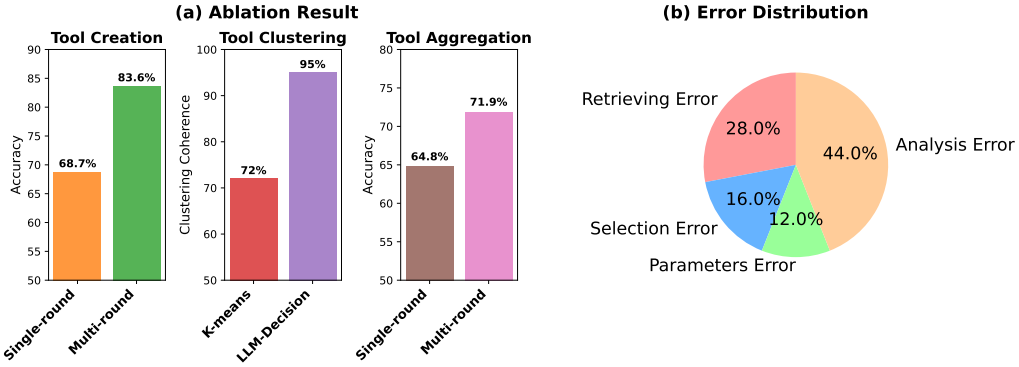

Figure 4: (a) Ablation results showing the effectiveness of our design; (b) Error distribution of GPT-4.1 in reasoning augmented by tools generated by TOOLLIBGEN.

Condition	Qwen3.5-35B-A3B	GPT-5.4-mini
Baseline	22.3	32.5
Fragmented skills	19.4	29.6
Aggregated skill	**33.7**	**40.2**

Table 2: Accuracy (%) on BrowseComp-Plus subset (sampled$N$=300).

such as query reformulation and multi-source verification, from 93k successful deep research trajectories (Li et al., 2026), and aggregate them into 2.4k higher-level skills using our framework. Since natural-language skills lack executable semantics, we adapt the reviewing criterion from answer-based execution checking to trajectory-prefix consistency: at the position where an atomic skill would be invoked, we compare whether conditioning the solver on the atomic skill or the aggregated skill leads to equivalent next-step plans. This preserves the central idea that an aggregated unit should induce behavior equivalent to the original unit it replaces.

We evaluate the resulting skill library on BrowseComp-Plus (Chen et al., 2025) with two solver models. As shown in Table 2, fragmented atomic skills underperform the baseline by 2–3 points, while our aggregated skills improve accuracy by $+11.4\%$ and $+7.7\%$ on Qwen3.5-35B-A3B and GPT-5.4-mini, respectively. These results suggest that the core idea of TOOLLIBGEN, cluster-level aggregation with behavioral verification, is not limited to Python code but can also apply to less structured natural-language skills. The full pipeline is provided in Appendix E.

5 Conclusion

In this work, we introduced TOOLLIBGEN, a pipeline for transforming a fragmented toolset into a structured Python library through clustering and a multi-agent refactoring aggregation. Our experiments across multiple domains and foundation models demonstrate that the aggregated library design significantly improves task accuracy compared with the fragmented toolset. This highlights the importance of shifting the focus from isolated tool generation to systematic organization and abstraction, which better supports scalable and reliable tool-augmented reasoning. Our study highlights how data can be more effectively leveraged to enhance downstream reasoning performance without additional training.

6 Limitations

Our approach has two main limitations. First, the tool generation process relies exclusively on successful CoT processes, ignoring valuable negative signals from failed attempts that could further improve tool robustness. Second, the resulting tool library is currently static and lacks a dynamic updating mechanism to revise or deprecate tools as new scenarios emerge. And the library also has a risk that automatically generated or retrieved tools may be incorrect, incomplete, or misused by the solver. We leave the integration of negative feedback and continuous library maintenance to future work.

References

Akhiad Bercovich and 1 others. 2025. Llama-nemotron: Efficient reasoning models. *Preprint*, arXiv:2505.00949.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. 2022. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*.

Zijian Chen, Xueguang Ma, Shengyao Zhuang, Ping Nie, Kai Zou, Andrew Liu, Joshua Green, Kshama Patel, Ruoxi Meng, Mingyi Su, and 1 others. 2025. Browsecomp-plus: A more fair and transparent evaluation benchmark of deep-research agent. *arXiv preprint arXiv:2508.06600*.

Aniket Didolkar, Anirudh Goyal, Nan Rosemary Ke, Siyuan Guo, Michal Valko, Timothy Lillicrap, Danilo Jimenez Rezende, Yoshua Bengio, Michael C Mozer, and Sanjeev Arora. 2024. Metacognitive capabilities of llms: An exploration in mathematical problem solving. *Advances in Neural Information Processing Systems*, 37:19783–19812.

Xinrun Du, Yifan Yao, Kaijing Ma, Bingli Wang, Tianyu Zheng, King Zhu, Minghao Liu, Yiming Liang, Xiaolong Jin, Zhenlin Wei, and 1 others. 2025. Supergpqa: Scaling llm evaluation across 285 graduate disciplines. *arXiv preprint arXiv:2502.14739*.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models. In *International Conference on Machine Learning*, pages 10764–10799. PMLR.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, and 1 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Zhiwei He, Tian Liang, Jiahao Xu, Qiuzhi Liu, Xingyu Chen, Yue Wang, Linfeng Song, Dian Yu, Zhenwen Liang, Wenxuan Wang, and 1 others. 2025. Deepmath-103k: A large-scale, challenging, decontaminated, and verifiable mathematical dataset for advancing reasoning. *arXiv preprint arXiv:2504.11456*.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213.

Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, and 1 others. 2022. Solving quantitative reasoning problems with language models. *Advances in neural information processing systems*, 35:3843–3857.

Chengpeng Li, Mingfeng Xue, Zhenru Zhang, Jiaxi Yang, Beichen Zhang, Xiang Wang, Bowen Yu, Binyuan Hui, Junyang Lin, and Dayiheng Liu. 2025. Start: Self-taught reasoner with tools. *arXiv preprint arXiv:2503.04625*.

Hongyu Li, Liang Ding, Meng Fang, and Dacheng Tao. 2024. Revisiting catastrophic forgetting in large language model tuning. In *Findings of the association for computational linguistics: EMNLP 2024*, pages 4297–4308.

Zhuofeng Li, Dongfu Jiang, Xueguang Ma, Haoxiang Zhang, Ping Nie, Yuyu Zhang, Kai Zou, Jianwen Xie, Yu Zhang, and Wenhu Chen. 2026. Openresearcher: A fully open pipeline for long-horizon deep research trajectory synthesis. *arXiv preprint arXiv:2603.20278*.

Jiaheng Liu, Dawei Zhu, Zhiqi Bai, Yancheng He, Huanxuan Liao, Haoran Que, Zekun Wang, Chenchen Zhang, Ge Zhang, Jiebin Zhang, and 1 others. 2025. A comprehensive survey on long context language modeling. *arXiv preprint arXiv:2503.17407*.

Yubo Ma, Zhibin Gou, Junheng Hao, Ruochen Xu, Shuohang Wang, Liangming Pan, Yujiu Yang, Yixin Cao, and Aixin Sun. 2024. SciAgent: Tool-augmented language models for scientific reasoning. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 15701–15736, Miami, Florida, USA. Association for Computational Linguistics.

Zhiyuan Ma, Zhenya Huang, Jiayu Liu, Minmao Wang, Hongke Zhao, and Xin Li. 2025. Automated creation of reusable and diverse toolsets for enhancing llm reasoning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 24821–24830.

Rui Meng, Ye Liu, Shafiq Rayhan Joty, Caiming Xiong, Yingbo Zhou, and Semih Yavuz. 2024. Sfr-embedding-mistral:enhance text retrieval with transfer learning. Salesforce AI Research Blog.

Grégoire Mialon, Roberto Dessì, Maria Lomeli, Christoforos Nalmpantis, Razvan Pascanu, Roberta Raileanu, Baptiste Rozière, Timo Schick, Jane Dwivedi-Yu, Asli Celikyilmaz, and 1 others. 2023. Augmented language models: a survey. *Transactions on Machine Learning Research*.

OpenAI. 2025a. Gpt-4.1.

OpenAI. 2025b. Gpt-5.

OpenAI. 2025c. gpt-oss model card. *Preprint*, arXiv:2508.10925.

Xiangyu Qi, Yi Zeng, Tinghao Xie, Pin-Yu Chen, Ruoxi Jia, Prateek Mittal, and Peter Henderson. 2023. Fine-tuning aligned language models compromises safety, even when users do not intend to! *arXiv preprint arXiv:2310.03693*.

Cheng Qian, Chi Han, Yi Fung, Yujia Qin, Zhiyuan Liu, and Heng Ji. 2023. CREATOR: Tool creation for disentangling abstract and concrete reasoning of large language models. In *The 2023 Conference on Empirical Methods in Natural Language Processing*.

Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, Jun Xu, and Ji-Rong Wen. 2025. Tool learning with large language models: A survey. *Frontiers of Computer Science*, 19(8):198343.

Elias Stengel-Eskin, Archiki Prasad, and Mohit Bansal. 2024. ReGAL: Refactoring programs to discover generalizable abstractions. In *International Conference on Machine Learning*.

Yu Sun, Xingyu Qian, Weiwen Xu, Hao Zhang, Chenghao Xiao, Long Li, Yu Rong, Wenbing Huang, Qifeng Bai, and Tingyang Xu. 2025. Reasonmed: A 370k multi-agent generated dataset for advancing medical reasoning. *arXiv preprint arXiv:2506.09513*.

Zhiruo Wang, Daniel Fried, and Graham Neubig. 2024. Trove: Inducing verifiable and efficient toolboxes for solving programmatic tasks. *arXiv preprint arXiv:2401.12869*.

Zora Zhiruo Wang, Apurva Gandhi, Graham Neubig, and Daniel Fried. 2025. Inducing programmatic skills for agentic tasks. In *Conference on Language Modeling (COLM)*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Binfeng Xu, Xukun Liu, Hua Shen, Zeyu Han, Yuhan Li, Murong Yue, Zhiyuan Peng, Yuchen Liu, Ziyu Yao, and Dongkuan Xu. 2023. Gentopia: A collaborative platform for tool-augmented llms. *arXiv preprint arXiv:2308.04030*.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Lifan Yuan, Yangyi Chen, Xingyao Wang, Yi R Fung, Hao Peng, and Heng Ji. 2023. Craft: Customizing llms by creating and retrieving from specialized toolsets. *arXiv preprint arXiv:2309.17428*.

Murong Yue, Wenlin Yao, Haitao Mi, Dian Yu, Ziyu Yao, and Dong Yu. 2024. Dots: Learning to reason dynamically in llms via optimal reasoning trajectories search. *arXiv preprint arXiv:2410.03864*.

Murong Yue, Jie Zhao, Min Zhang, Liang Du, and Ziyu Yao. 2023. Large language model cascades with mixture of thoughts representations for cost-efficient reasoning. *arXiv preprint arXiv:2310.03094*.

Min Zhang, Jianfeng He, Shuo Lei, Murong Yue, Linhan Wang, and Chang-Tien Lu. 2024. Can llm find the green circle? investigation and human-guided tool manipulation for compositional generalization. In *ICASSP 2024-2024 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 11996–12000. IEEE.

Shengyu Zhang, Linfeng Dong, Xiaoya Li, Sen Zhang, Xiaofei Sun, Shuhe Wang, Jiwei Li, Runyi Hu, Tianwei Zhang, Guoyin Wang, and 1 others. 2026. Instruction tuning for large language models: A survey. *ACM Computing Surveys*, 58(7):1–36.

Boyuan Zheng, Michael Y Fatemi, Xiaolong Jin, Zora Zhiruo Wang, Apurva Gandhi, Yueqi Song, Yu Gu, Jayanth Srinivasa, Gaowen Liu, Graham Neubig, and 1 others. 2025. SkillWeaver: Web agents can self-improve by discovering and honing skills. *arXiv preprint arXiv:2504.07079*.

A Algorithm

The algorithm of our proposed method is shown in Algorithm 1.

B Experiment Details

B.1 Dataset

For the Nemotron dataset, we first performed data cleaning, which included removing duplicate questions and using GPT-5 to evaluate question quality in order to filter out under-specified problems. In terms of topic filtering, we only retained questions related to physics, chemistry, and biology. Since Nemotron does not guarantee the correctness of the original answers, we re-answered the questions using GPT-5 and kept only those whose answers were consistent with the original ones. For DeepMath-103k, we used all 103k questions. For ReasonMed, we randomly sampled 100k questions. Since the correctness of the original answers could not be guaranteed, we applied the same strategy as with Nemotron: GPT-5 was used to re-answer the questions, and only those consistent with the original answers were retained. The number of questions, tools of fragmented toolset and tools of our library are shown in Table 3.

During the testing phase, for the seen cases, most questions from the Nemotron and ReasonMed training sets were relatively straightforward, allowing the LLM to answer them directly without tools. This made it difficult to effectively compare the performance of different tool aggregation strategies, as variations in strategy had little impact on the final outcome. To address this, we first performed CoT inference on all cases. From the cases where the CoT inference failed, we randomly sampled 1k questions as hard questions from each dataset. It is important to note that these are multiple-choice questions; when solving these hard examples, the LLM might have made a wrong choice between two plausible options, leading to the first time CoT inference failure. In subsequent sampling, the LLM might happen to guess the correct option, which is why the CoT success rate isn't zero. Given the inherently high difficulty of the DeepMath dataset, we directly and randomly sampled 1k questions from it for our evaluation. For superGPQA, we use the questions with labels from physics, chemistry, and biology as science QA, math-labeled questions were used as math QA, and medicine-labeled questions were used as medical QA.

B.2 Hyper-parameters

When we perform clustering, we initially set the number of seed tools to 1k. We update in batches of 200, with a maximum of 500 update rounds. For the creation and aggregation phases, the maximum number of modification rounds is set to 3.

Dataset and Pipeline The experiment was conducted on the physics domain of the MegaScience dataset. We utilized about 28k physical queries and CoT processes to generate atomic, query-specific skills. The generation and clustering procedures strictly followed the same methodology used in our main experiments. During the aggregation phase, instead of merging functions into a single Python library, the LLM was tasked with synthesizing the clustered atomic skills into multiple high-level, general-purpose skill packages conforming to the aforementioned schema.

Evaluation Setup and Results. We evaluated the generated skills on a randomly held-out set of 500 MegaScience physics questions. For the evaluation solver, we employed the agent capable of digesting the entire package schema, retrieving related skills, reading the `skill_md` for instructions, and executing the provided `scripts` within its environment.

C Ablation Study Setting

To understand the contribution of each component of our framework, we conduct a thorough ablation study for each key component of our pipeline.

Question-specific Tool Creation We first evaluated the necessity of our multi-round generation process, which incorporates validation from an LLM_{solver}. We compared this approach against a single-pass generation on a randomly sampled 300 questions from the subset of Nemotron used in testing, using Qwen3-8B as the LLM_{solver}. Performance was measured by the accuracy of the generated tool in solving the corresponding question. The results show that the multi-round process yields higher accuracy, confirming that iterative validation and refinement are crucial for creating effective tools.

Tool Clustering Next, we compared the quality of our LLM-based clustering method against a standard K-means baseline. For the baseline, we applied K-means to SFR-embeddings derived from each tool's title and description, setting the number of clusters (K) to be identical to the number

Algorithm 1 Algorithm of TOOLLIBGEN

Input:
Dataset $\mathcal{D} = \{(Q_i, CoT_i)\}_{i=1}^{N}$
Output: Python library $\mathcal{L}$

$\mathcal{T}, \mathcal{L}, \mathcal{C}$, SourceQuestionMap, SourceQuestionMapForCluster $\leftarrow \emptyset, \emptyset, \emptyset, \{\}, \{\}$
▷ **Phase 1: Question-specific Tool Creation**
for each $(Q_i, CoT_i) \in \mathcal{D}$ **do**
  $\mathcal{T}_i \leftarrow LLM(\text{“Abstract”}, Q_i, CoT_i)$
  for each tool t in $\mathcal{T}_i$ **do**
    while $turn <$ Max_Checking_Turns **do**
      Trajectory $\leftarrow LLM_{solver}(\text{“Solve the question”}, Q_i, t)$ ▷ Solve with LLM_{solver}
      Decision $\leftarrow$ LLM$(Q_i, CoT_i, t,$ Trajectory$)$
      if Decision is Pass **then break**
      $t \leftarrow LLM(\text{“Refine”}, t, \text{Trajectory})$
    $\mathcal{T} \leftarrow \mathcal{T} \cup \{t\}$; SourceQuestionMap$[t] \leftarrow (Q_i, CoT_i)$
▷ **Phase 2: Tool Clustering**
$\mathcal{T}_{\text{seed}} \leftarrow$ RandomSample$(\mathcal{T}, m)$; $\mathcal{T}_{\text{rem}} \leftarrow \mathcal{T} \setminus \mathcal{T}_{\text{seed}}$
$\{label_k\}_{k=1}^{K} \leftarrow LLM(\text{“Propose Clusters”}, \mathcal{T}_{\text{seed}})$
for each batch $\mathcal{T}_b$ of size b in $\mathcal{T}_{\text{rem}}$ **do**
  $\{label_k\}_{k=1}^{K} \leftarrow LLM(\text{“Update Clusters”}, \{label_k\}_{k=1}^{K}, \mathcal{T}_b)$ ▷ K may change
$\mathcal{C} \leftarrow \{C_k\}_{k=1}^{K}$, each $C_k \leftarrow \emptyset$ from $\{label_k\}_{k=1}^{K}$
for $t \in \mathcal{T}$ **do**
  $k \leftarrow LLM(\text{“Assign tools to clusters”}, \{label_k\}_{k=1}^{K}, t)$
  $C_k \leftarrow C_k \cup \{t\}$
  SourceQuestionMapForCluster$[C_k].add($SourceQuestionMap$[t])$
▷ **Phase 3: Tool Aggregation**
for $C_k \in \{C_1, \ldots, C_K\}$, where $C_k = \{t_{km}\}_{m=1}^{m_k}$ **do**
  Blueprint $\leftarrow$ Code Agent(“Design blueprint”, C_k)
  $C_k^* \leftarrow$ Code Agent(“Implement codes”, C_k, Blueprint)
  while $turn <$ Max_Checking_Turns **do**
    all_feedback $\leftarrow$ Reviewing Agent(C_k^*, SourceQuestionMapForCluster$[C_k]$) ▷ Solve with LLM_{solver}
    if AllFeedbacksShowingPass(all_feedback) **then break**
    $C_k^* \leftarrow LLM(\text{“Refine codes”}, C_k, \text{all_feedback})$
  $\mathcal{L} \leftarrow \mathcal{L} \cup \{C_k^*\}$
return $\mathcal{L}$

	# of Question	# of Question-Specific Tool	# of Library Tool
Science	33k	48k	3.1k
Math	103k	175k	8.9k
Medical	100k	142k	5.8k

Table 3: The number of toolset of different domains

of leaf nodes generated by our method. To evaluate, we randomly sampled 100 tools and retrieved three other tools from the same cluster using both techniques. A clustering case was deemed correct only if manual inspection confirmed that all three retrieved tools belonged to the same functional sub-domain. Our method achieved 95% accuracy, significantly outperforming the K-means baseline, which only reached 72%, demonstrating its superior ability to create coherent tool groups.

Tool Aggregation Finally, for tool aggregation, we similarly validated the effectiveness of the multi-round generation and validation process. Following the same experimental setup as in the tool creation study, we observed that a single-pass approach is often insufficient, as the LLM fails to abstract all reasoning scenarios covered by the tools, leading to critical omissions. The multi-turn revision process proved essential for significantly improving the aggregated library's final performance.

D Can we teach an LLM to propose a better search query via supervised-finetuning?

We explore whether we could improve an LLM's tool-augmented reasoning by teaching it to ask a better search query during retrieval. We designed a data collection method using 5k randomly sampled examples from the Nemotron dataset's science domain (excluding the testing data used in

Table 1) to obtain higher-quality search queries. For each tool t derived from a question Q, we trace the aggregated tool t^* and pair the question Q with the aggregated tool t^*. Subsequently, we prompt Qwen3-8B to roll out different searching queries based on question Q and only keep the query that can correctly recall t^*. Subsequently, we fine-tune Qwen3-8B with the correctly retrieved trajectory. After fine-tuning, the accuracy of Qwen3-8B on the Nemotron dataset reached 72.8%, a slight improvement over the 71.9% accuracy without this training (as shown in Table 1). We believe this marginal gain is because the LLM struggles to learn the nuances of effective query formulation through supervised-finetuning alone, as most queries it generates appear reasonable even without training. Future work could benefit from introducing hard negative examples to help the model better distinguish the nuanced difference between queries.

E Natural Language Skill Aggregation Pipeline

This appendix details how we adapt TOOLLIBGEN to the aggregation of natural language skills. The pipeline strictly follows the three-stage structure of the main text (creation, clustering, aggregation); the adaptation is confined to the verification mechanism.

E.1 Data Source and Atomic Skill Extraction

We extract atomic skills from 93,508 successful deep research trajectories. Each trajectory consists of a research query and the multi-step reasoning and retrieval process the agent executed to answer it. For each trajectory, we use an LLM to identify the strategic steps that played a key role in the research and format them as atomic skills, each containing three fields: **title**, **trigger condition**, and **procedure**.

Crucially, every atomic skill is associated with an implicit verification sample. The source trajectory provides (i) the original question, (ii) the trajectory position (cutoff point) where the skill should be invoked, (iii) the context (prefix) before this position, and (iv) the research behavior actually taken afterwards. This association is critical for the subsequent verification stage—it allows us to retain TOOLLIBGEN's behavioral-equivalence criterion without relying on code execution.

The extraction yields **268,764 atomic skill instances**. We apply a lightweight textual normalization on titles (lemmatization, stopword removal) as preprocessing, yielding 141,787 normalized-title skills.

E.2 Clustering

Clustering follows the propose-then-assign procedure of main text. Specifically: (i) we randomly sample a seed subset and prompt the LLM to propose a tree-structured cluster hierarchy, retaining the leaf nodes; (ii) the remaining skills are processed in batches, with the LLM iteratively updating the cluster tree and assigning skills to leaf nodes. This yields **312 leaf clusters** covering all 268,764 atomic skill instances.

E.3 Aggregation: Multi-agent Framework with Trajectory-prefix Consistency

Aggregation follows the Coding–Reviewing agent framework; the only adaptation is in the Reviewing agent's equivalence verification mechanism.

E.3.1 Coding Agent

For each cluster, the Coding agent receives *all* member skills (not a sample) and first generates a blueprint that compresses the cluster's functionality into a set of **behavior buckets**—each bucket representing a reusable research behavior branch, such as:

- query reformulation after failed search
- entity disambiguation with multiple constraints
- in-document keyword drilldown
- multi-source verification
- source access fallback

Behavior bucketing compresses thousands of atomic skills into a structured functional summary that fits within the LLM context, without losing functional diversity.

The Coding agent then generates multiple aggregated skills based on the blueprint. We do not require each cluster to produce a single aggregated skill: when behavioral diversity within a cluster is large, forcing a single skill makes the result overly generic and harms downstream performance.

E.3.2 Reviewing Agent: Trajectory-prefix Consistency Check

This is the core adaptation point. In the main text, the Reviewing agent verifies functional preservation by having $\text{LLM}_{\text{solver}}$ solve the original question and comparing the answer. In the skill setting, **we replace this criterion with a trajectory-prefix consistency check, while the underlying logic remains identical**: the aggregated skill should be able to induce a functionally equivalent next-step behavior in the solver at the position where the atomic skill was originally invoked.

Concretely, for each atomic skill in a cluster, we locate the cutoff point in its source trajectory and construct two conditions:

- **Atomic condition.** Input `(question, trajectory prefix, atomic skill)` to the solver, and ask it to output the next research plan. This plan serves as the **reference atomic plan**.

- **Aggregated condition.** Input `(question, trajectory prefix, candidate aggregated skill)` to the solver, and ask it to output the next plan.

The Reviewing agent acts as a judge and compares whether the two plans are **functionally consistent**. The criterion is not literal match but **functional containment**: whether the aggregated plan contains the core research logic of the atomic plan.

For example, if the atomic plan is:

> *The current search failed to locate the target person. The next step should relax over-specific constraints, first using the person's role and affiliation as anchors to retrieve candidates, then verifying year and award one by one.*

The aggregated plan passes only if it contains all of the following: recognizing the current state as search failure / ambiguous entity; relaxing over-specific queries; using stable entity anchors; retrieving candidates first and verifying remaining constraints afterwards; not guessing the answer directly. If the aggregated plan degenerates into generic phrasing like "try another search", it is treated as failing— it fails to preserve the specific operational logic of the atomic skill.

E.3.3 Iterative Repair

If the consistency check fails, the Reviewing agent outputs the specific missing behaviors, e.g.:

```
Missing behavior: The aggregated
skill does not specify opening
a promising document and using
find() within the document to
locate date/title constraints.
```

The Coding agent then performs **targeted repair**, modifying only the missing behavior branch rather than regenerating from scratch. After repair, the consistency check is rerun. This loop runs for at most 3 iterations; if some atomic skills remain uncovered after 3 rounds, they are retained as singleton skills to guarantee functional preservation.

E.3.4 Two-tier Verification for Scalability

At the scale of 312 clusters and 268k atomic skill instances, running the full trajectory-prefix consistency check on every atomic skill is computationally prohibitive. We therefore adopt a **two-tier verification strategy**:

- **Tier 1 (scalable bucket audit).** Check whether the aggregated skill explicitly covers the trigger, operation, and fallback of each behavior bucket. Uncovered buckets enter the repair loop.

- **Tier 2 (trajectory-prefix consistency validation).** Sample from high-weight buckets, long-tail buckets, and high-risk buckets after Tier-1 repair, and run the full trajectory-prefix consistency check.

The relationship between the two tiers is: the bucket audit is a scalable proxy that makes full-cluster aggregation feasible at scale, while trajectory-prefix consistency is the actual criterion that defines functional preservation— consistent with the philosophy of TOOLLIBGEN in the main text. The aggregation produces **2,398 aggregated skills** across 312 clusters.

E.4 Skill Usage at Inference Time

To prevent skills from being treated as factual sources by the agent, we strictly design the skill library as a **strategy layer**:

- The agent accesses skills only through `skill_search` and `open_skill` interfaces.

- Final answers must be supported by evidence returned from the corpus's `search` / `open` / `find` interfaces.
- Skills guide retrieval strategy only and do not enter the answer's citation chain.

In addition, we design a **selector based on observable signals** for fair comparison. The selector does not access gold answers and makes routing decisions based only on agent-reported state signals (whether the answer is marked insufficient, citation count, opened-document count, etc.). **The selector uses an identical framework across baseline, fragmented, and aggregated conditions, with only the underlying skill library differing**, ensuring fair comparison.

E.5 Correspondence with TOOLLIBGEN in the Main Text

The core structure of the pipeline—*cluster-level global aggregation + multi-agent iterative refinement + behavioral equivalence verification*—is identical across both settings. The adaptation is confined to the underlying verification signal: in settings lacking executable semantics, we replace "executing the original question" with "comparing solver behavior at the original trajectory position". This substitution preserves the essence of the functional equivalence criterion—**the aggregated tool should induce equivalent behavior at the position where the original atomic tool would be invoked**—while making the method applicable to less structured forms of knowledge.

F Example

Question 1:
A particle moves along a straight line with its velocity as a function of time given by $v(t) = 5t^2 + 2t$. What is its acceleration function, a(t)?

Tool 1:

```
def apply_bayes(likelihood_A: float, prior_A:
    float,
likelihood_not_A: float, prior_not_A: float) ->
    float:
    # Calculate the numerator of Bayes' theorem:
        P(B|A) * P(A)
    numerator = likelihood_A * prior_A

    # Calculate the total probability of the
        evidence B (the denominator):
    # P(B) = P(B|A)P(A) + P(B|not A)P(not A)
    marginal_likelihood = (likelihood_A * prior_A
        )
    \+ (likelihood_not_A * prior_not_A)

    # Avoid division by zero if the evidence is
        impossible
    if marginal_likelihood == 0:
        return 0.0

    # Compute the posterior probability
    posterior_A = numerator / marginal_likelihood

    return posterior_A
```

Question 2:
A mother with blood type O and a father with blood type AB have twins, both sons, with blood type B. Given that 32% of all twins have different genders, calculate the probability that the twins are identical.

Tool 2:

```
from collections import Counter
import json

def get_abo_offspring_allele_distribution
(mother_genotype: str, father_genotype: str) ->
    dict:
    # Get alleles from each parent
    mother_alleles = list(mother_genotype)
    father_alleles = list(father_genotype)

    # Generate all possible offspring allele
        combinations
    possible_genotypes = []
    for m_allele in mother_alleles:
        for f_allele in father_alleles:
            # Sort to standardize (e.g., AO not
                OA)
            genotype = ''.join(sorted((m_allele,
                f_allele)))
            possible_genotypes.append(genotype)

    # Count occurrences of each genotype
    genotype_counts = Counter(possible_genotypes)

    # Calculate probabilities
    total_combinations = len(possible_genotypes)
    genotype_distribution = {geno: count /
        total_combinations
    for geno, count in genotype_counts.items()}

    return genotype_distribution
```

Question 3:
You have two coins: a fair coin with a probability of heads as 0.5, and a biased coin with a probability of heads as 0.8. You randomly select one of these coins and flip it repeatedly, obtaining heads each time. Determine the minimum number of consecutive heads required such that the probability of the coin being fair is less than 0.1.

Tool 3:

```
def equal_priors(items):
    """
    Assign equal prior probability to each item
        in the list.
```

Stage	Python Tool Pipeline (Main)	NL Skill Pipeline (This Appendix)
Extraction	Python functions from CoT	(title, trigger, procedure) from trajectory
Clustering	Propose-then-assign hierarchical	**Identical**
Aggregation framework	Coding + Reviewing agent loop	**Identical**
Equivalence criterion	$\text{LLM}_{\text{solver}}$ solves original Q	Trajectory-prefix consistency
Scalable proxy	Not needed	Bucket-level coverage audit

Table 4: Correspondence between the Python tool pipeline (main text) and the natural language skill pipeline (this appendix).

```
    Returns a dict mapping item -> 1/len(items).
    """
    n = len(items)
    if n == 0:
        return {}
    p = 1.0 / n
    return {item: p for item in items}

def apply_bayes(likelihood_A: float, prior_A:
     float,
                likelihood_not_A: float,
                    prior_not_A: float) -> float:
    """
    Returns P(A|B) using Bayes' theorem for a
         binary hypothesis.
    """
    num = likelihood_A * prior_A
    denom = num + likelihood_not_A * prior_not_A
    if denom == 0:
        raise ZeroDivisionError("Total
             probability is zero; check inputs.")
    return num / denom
```

Aggregated Tool:

```
from typing import List, Dict, Any, Union
import math
from abc import ABC, abstractmethod

class Hypothesis:
    """Represents a single hypothesis with a name
         and a prior probability."""
    def __init__(self, name: str, prior: float):
        if not (0.0 <= prior <= 1.0):
            raise ValueError("Prior probability
                 must be between 0 and 1.")
        self.name = name
        self.prior = prior

    def __repr__(self) -> str:
        return f"Hypothesis(name='{self.name}',
             prior={self.prior:.4f})"

class HypothesisSet:
    """Manages a set of mutually exclusive
         hypotheses, ensuring priors sum to 1."""
    def __init__(self, hypotheses: List[
         Hypothesis]):
        total_prior = sum(h.prior for h in
             hypotheses)
        if not math.isclose(total_prior, 1.0,
             rel_tol=1e-9):
            raise ValueError(f"Priors must sum to
                 1, but they sum to {total_prior}
                ")
        self._hypotheses = {h.name: h for h in
             hypotheses}

    def get(self, name: str) -> Hypothesis:
        return self._hypotheses[name]

    @property
    def names(self) -> List[str]:
        return list(self._hypotheses.keys())

    def __iter__(self):
        return iter(self._hypotheses.values())

class BaseEvidence(ABC):
    """Abstract base class for all forms of
         evidence."""
    @abstractmethod
    def get_likelihoods(self, hypothesis_set:
         HypothesisSet) -> Dict[str, float]:
        """
        Returns a dictionary mapping each
             hypothesis name to its likelihood
        P(Evidence | Hypothesis).
        """
        pass

class DirectLikelihoodEvidence(BaseEvidence):
    """
    Evidence defined by direct likelihood values
         (can be probabilities or densities).
    This class solves the problem of densities >
         1.
    """
    def __init__(self, likelihood_map: Dict[str,
         float]):
        for name, val in likelihood_map.items():
            if val < 0:
                raise ValueError(f"Likelihood for
                     '{name}' cannot be negative.")

        self._likelihood_map = likelihood_map

    def get_likelihoods(self, hypothesis_set:
         HypothesisSet) -> Dict[str, float]:
        if not set(self._likelihood_map.keys())
             == set(hypothesis_set.names):
            raise ValueError("Likelihood map must
                 contain keys for all hypotheses.
                ")
        return self._likelihood_map

class RepeatedSuccessEvidence(BaseEvidence):
    """
    Evidence from 'n' consecutive successes,
         where the success probability
    depends on the hypothesis.
    """
    def __init__(self, n_successes: int,
```

```python
        success_prob_map: Dict[str, float]):
        if n_successes < 0:
            raise ValueError("Number of successes cannot be negative.")
        self.n_successes = n_successes
        self.success_prob_map = success_prob_map

    def get_likelihoods(self, hypothesis_set: HypothesisSet) -> Dict[str, float]:
        likelihoods = {}
        for name in hypothesis_set.names:
            p = self.success_prob_map[name]
            likelihoods[name] = p ** self.n_successes
        return likelihoods

class BayesianSolver:
    """The engine that calculates posterior probabilities from hypotheses and evidence."""
    def __init__(self, hypothesis_set: HypothesisSet):
        self.hypothesis_set = hypothesis_set
        self.evidence_list: List[BaseEvidence] = []

    def add_evidence(self, evidence: BaseEvidence):
        self.evidence_list.append(evidence)

    def calculate_posterior(self) -> Dict[str, float]:
        """
        Calculates the posterior probabilities using log-space for numerical stability.
        """
        # Start with log of priors
        log_posteriors = {h.name: math.log(h.prior) for h in self.hypothesis_set}

        # Accumulate log likelihoods from all evidence
        for evidence in self.evidence_list:
            likelihoods = evidence.get_likelihoods(self.hypothesis_set)
            for name in self.hypothesis_set.names:

                # Use a small epsilon to avoid log(0)
                log_posteriors[name] += math.log(likelihoods[name] + 1e-300)

        # Normalize to get posterior probabilities
        # (This is a stable way to convert log probabilities back to a distribution)
        max_log_p = max(log_posteriors.values())
        posteriors_unnormalized = {name: math.exp(log_p - max_log_p) for name, log_p in log_posteriors.items()}
        total_p = sum(posteriors_unnormalized.values())

        if total_p == 0: # Handle case where all probabilities are zero
            num_hypotheses = len(self.hypothesis_set.names)
            return {name: 1.0 / num_hypotheses for name in self.hypothesis_set.names}

        final_posteriors = {name: p / total_p for name, p in posteriors_unnormalized.items()}
        return final_posteriors


def bayesian_scenario_solver(scenario_spec_json: str) -> Dict[str, Any]:
    """
    A high-level facade to model and solve various Bayesian scenarios from a JSON string.

    Args:
        scenario_spec_json (str): A JSON string describing the entire problem.
            - 'hypotheses': List of dicts, e.g., [{'name': 'H1', 'prior': 0.5}, ...]
            - 'evidence': (Optional) List of evidence dicts. Each needs a 'type'.
            - 'goal': (Optional) A dict describing a complex goal, like finding a minimum 'n'.

    Returns:
        A dictionary containing the results of the analysis.
    """
    # 1. Parse the JSON string input into a Python dictionary
    try:
        scenario_spec = json.loads(scenario_spec_json)
    except json.JSONDecodeError as e:
        raise ValueError(f"Invalid JSON format: {e}")
    hyp_defs = [Hypothesis(h['name'], h['prior']) for h in scenario_spec['hypotheses']]
    hypothesis_set = HypothesisSet(hyp_defs)

    # 2. Check for a complex goal (like for Problem 3)
    if 'goal' in scenario_spec and scenario_spec['goal']['type'] == 'find_minimum_n':
        goal = scenario_spec['goal']
        n = 1
        while True:
            solver = BayesianSolver(hypothesis_set)
            evidence = RepeatedSuccessEvidence(
                n_successes=n,
                success_prob_map=goal['success_prob_map']
            )
            solver.add_evidence(evidence)
            posteriors = solver.calculate_posterior()

            target_posterior = posteriors[goal['hypothesis_to_track']]

            condition_met = False
            if goal['condition'] == '<' and target_posterior < goal['threshold']:
```

```
                condition_met = True
            elif goal['condition'] == '>' and
                 target_posterior > goal['
                 threshold']:
                condition_met = True

            if condition_met:
                return {
                    'status': 'Goal Reached',
                    'result': {'minimum_n': n},
                    'final_posteriors': posteriors
                }
            n += 1
            if n > 1000: # Safety break
                raise RuntimeError("Exceeded 1000
                     iterations without reaching
                     goal.")

    # 3. Standard posterior calculation (for
         Problems 1 & 2)
    solver = BayesianSolver(hypothesis_set)
    if 'evidence' in scenario_spec:
        for ev_spec in scenario_spec['evidence']:
            if ev_spec['type'] == '
                 direct_likelihoods':
                evidence =
                     DirectLikelihoodEvidence(
                     likelihood_map=ev_spec['
                     values'])
                solver.add_evidence(evidence)
            # Other evidence types could be added
                 here
            else:
                raise ValueError(f"Unknown
                     evidence type: {ev_spec['type
                     ']}")

    posteriors = solver.calculate_posterior()
    return {
        'status': 'Calculation Complete',
        'result': {'posteriors': posteriors}
    }
```

G Prompt

Tool Generation Prompt

You are an expert in creating tools. Below is the single worked example we have:
Question:
{question}
CoT Answer:
{CoT Process}
YOUR TASK:
Write one or multiple general tool that can solve similar sub question. Each tool must be an executable Python function designed to perform one atomic reasoning step.
Output Format:
For the question, generate one general tools, use:
- Each tool must be a executable Python function designed to the specific question.
- Make the Python function general: any database or tool should work with minimal tweaks. Only provide code for steps that truly benefit from automation.
- No code is needed for selecting the final answer; toolkits are for supporting intermediate reasoning only.
- Try to save knowledge into static variables; don't just think of the calculation process as a tool, but also think of the knowledge sentences as tools;
- For example, for the question: Can an object's momentum change without experiencing net acceleration? No calculation codes are needed, but knowledge sentences are needed for analysis. Even just some hints in the Python function is fine, but you should make sure that the knowledge is saved into static variables. - In this phase, the format is:

```
<tool1><code>
def function_name(param_1_name,
↪  param_2_name,...):
[Implement a general function for one
↪  sub-question]
return f"explain the result: {{result}}"
<\code><\tool1><tool2>...<\tool2>
```

Tool Verification Prompt

You are an expert problem solver who uses available tools to analyze and solve questions.
Question to solve:
{question}
Available Tools:
{tools description}
Your Task:
1. Use the available tools systematically to help analyze and solve the question;
2. Make sure to actually use the tools in your analysis process;
3. Provide clear reasoning for each step;
4. Give a final answer based on your tool-assisted analysis;
Final Response Format:

```
<analysis>
[Show tool outputs and explain how they
↪  contribute
to your analysis]</analysis>
<answer>Your conclusive answer</answer>
```

Tool Refinement Prompt

You are an expert tool effectiveness evaluator.
Original Question: {question}
Generated Tools: {tools}
Ground Truth Answer: {answer}
Tool Usage Analysis Results: {evaluation_results}
Your Task: Analyze whether the generated tools were truly effective for solving the question:
1. Tool Utility Assessment:
- Did the tools provide meaningful assistance in solving the question?
- Were the tools actually used in the solution process?
- Did the tools contribute to reaching the correct answer?
2. Tool Quality Assessment:
- Are the tools properly implemented and functional?
- Do the tools address the core reasoning steps needed for this type of question?
- Are the tools generalizable to similar questions?
3. Code Refinement:
Based on the effectiveness analysis, refine the tools to make them more useful for solving the question.
Response Format
Detailed explanation of why tools are effective or ineffective If ineffective, specific suggestions for improvement

```
<tool1><code>
```python
def function_name(param_1_name...):
[Implement a general function]
return f" explain the result: {{result}}."
</code></tool1><tool2>...</tool2>...
```

Cluster Proposing Prompt

You need to aggregate the following tools into a hierarchy. You do not need to set each tool in different nodes. In contrast, you should set the tools that are similar to each other in the same node. Please make sure that the hierarchy should not be too shallow. Deep hierarchy and detailed classification are preferred. The depth of the hierarchy should be {cluster_depth}.
The function_name of tools should NOT be the last layer leaf node of the hierarchy. Do not need to include the function_name in the hierarchy. tools:
{tool_lst}
Your output should be a JSON object with a hierarchical structure.

```
{{
  "clusters": [ {{id, level, parent,
  ↪  children}} . . . ],
  }}
example:
{{
  "clusters": [
    {{
      "id": "c_root",
      "level": 0,
      "parent": null,
      "children": ["c_math", "c_utils",...]
    }},
    {{
      "id": "c_math",
      "level": 1,
      "parent": "c_root",
      "children": ["c_arith", "c_stat",...]
    }},
    {{
      "id": "c_linear_algebra",
      "level": 2,
      "parent": "c_math",
      "children": ["c_matrix","c_singular_v
      ↪  alue_decomposition",...],
    }},
    {{
      "id": "c_stat",
      "level": 2,
      "parent": "c_math",
      "children": ["c_stat_mean",
      ↪  "c_stat_t_test",...],
    }},
  ]
}}
```

DELIVERABLE Return ONLY the JSON array described in OUTPUT FORMAT. No any other text. No “‘json.

Cluster Update Prompt

Given the current hierarchy and new tools to integrate, generate specific operations to update the hierarchy incrementally instead of rewriting the entire structure.
Analyze the new tools and determine what changes are needed:
1. ADD_NODE: Create new clusters for tools that don’t fit existing categories
2. MODIFY_NODE: Update existing cluster properties if needed
3. No operations if tools fit perfectly into existing leaf nodes
Current hierarchy:
current_hierarchy
New tools to integrate:
tool_lst
Your output should be a JSON object with specific operations:

```
{{
  "operations": [
    {{
      "action": "ADD_NODE",
      "node_id": "new_cluster_id",
      "level": ...,
      "parent": "parent_cluster_id",
      "description": "Description of what
      ↪  this cluster represents",
      "reasoning": "Why this new cluster is
      ↪  needed for the new tools"
    }},
    {{
      "action": "MODIFY_NODE",
      "node_id": "existing_cluster_id",
      "changes": {{
        "add_children": ["new_child_id1",
        ↪  "new_child_id2"]
      }},
      "reasoning": "Why this modification is
      ↪  needed"
    }}
  ]
}}
```

Guidelines:
```

- Only create new nodes when new tools represent significantly different functionality
- Prefer adding to existing leaf nodes when tools are similar enough
- Maintain proper parent-child relationships and level consistency
- Keep the hierarchy depth appropriate (not too shallow, not too deep)
- Provide clear reasoning for each operation

DELIVERABLE Return ONLY the JSON array described in OUTPUT FORMAT. No any other text. No “‘json.

Blueprint Design Prompt

Persona:
You are a Senior Python Library Architect; you transform fragmented helper functions into coherent, maintainable knowledge-libraries by applying rigorous knowledge-engineering practice.
Your Mission:
You will receive a list of Python functions as discrete tools that all belong to the same sub-domain. Design a **Refactoring Blueprint** that reorganises those tools into a catalogue of **Static Inference Blocks (SIBs).**
Static Inference Block (SIB) Definition
A SIB is a reusable knowledge capsule that is composed of one or multiple Python classes to construct one specifc problem-solving scenario and multiple public function to wrap up all the functionality of the collected discrete tools. The def function can accept multiple parameters and return a multi-paragraph explanation string that follows the standard template shown below.
It should follow the following requirements: • accepts a well-defined set of input facts (pre-conditions)
• instantly infers deterministic outputs (formulae, numbers, long–form explanations)
• groups as many original tools as logically coherent—functionality must remain complete and loss-less.
Example of SIB:

```
# Given tool code 1:
def calculate_potential_energy(mass,
↪  height):
    Gravity = 9.81
    return mass * Gravity * height

# Given tool code 2:
def get_gravity(planet_name):
    if planet_name == "Earth":
        return 9.81
    elif planet_name == "Mars":
        return 3.71
    else:
        return 0

# The SIB should be composed of the
↪  following one class
and one public function:
class _PlanetaryPhysics:
    def __init__(self, planet_name):
        self.planet_name = planet_name
        _GRAVITY_MAP = {{
            "Earth": 9.81,
            "Mars": 3.71
        }}
        self.gravity =
        ↪  _GRAVITY_MAP[planet_name]
    def calculate_potential_energy(self,
    ↪  mass, height):
        return mass * self.gravity * height
    def get_gravity(self):
        return self.gravity

def calculate_potential_energy(mass, height,
↪  planet_name="Earth"):
    planetary_physics =
    ↪  _PlanetaryPhysics(planet_name)
    return planetary_physics.calculate_pote
    ↪  ntial_energy(mass,
    ↪  height)
```

Core Requirements 1. High Cohesion / Low Coupling: Group functions that operate on the same core data or represent the same conceptual scenario. SIBs should be self-contained and independent. 2. Scenario-Focused Classes: Each SIB Class should just model one specific, concrete scenario (e.g., _ProjectileMotion, _GasContainerState). The class is initialized with the base facts of the scenario, from which all other properties can be derived. But the scenario should be specific, not too general.
3. Lossless Abstraction: The public execute function must provide a way to access the full functionality of all original tools it covers. Use optional parameters to control which specific calculations are performed. But the parameters should be simple to understand and use.
4. Descriptive Naming: SIB titles and class names must be explicit and descriptive. Avoid vague, generic umbrella names such as PhysicsToolkit or MathHelpers.
5. Blueprint, Not Code: Your output is a design document (the blueprint), not the final Python code.
6. All parameters should be simple to understand and use. No any hard-encoded condition in the parameters, e.g., "if parameter == a, then use function1, else use function2,". In this case, you need to create more than one public function to cover all the scenarios rather than using hard-encoded condition.
Output Format: The Refactoring Blueprint
Your entire output must be a single markdown document. This document will contain multiple SIB descriptions (depending on how many scenario you can infer from the current tools), each strictly adhering to the following metadata template.
Mandatory SIB Metadata (document for *every* SIB): [SIB]Title [Description] (Provide a concise, high-level summary in plain language describing what this SIB is and what problem it solves.) [SIB Class Description] (Design one or more classes. Each class should model one scenario where, given a few initial inputs, many other properties can be deterministically calculated. Describe the purpose of each class and what initial state it will be constructed with. Output every class’s init function, inner function name, description and parameters.) [Public Function Description] (Describe one or more public functions. Detail which parameters it should accept to ensure all scenarios and functionalities of the covered tools are accessible. [Covered Tools] (List all the tool indices that are covered by this SIB, separated by commas. E.g., 1, 2, 3)
Input: All Python Function Tool Code: {tool_code_list}
Deliverable: Now start to write the blueprint. Your final output is the complete Refactoring Blueprint in markdown format. Do not include any Python implementation code. Exactly follow the format:

<SIB>

```
[SIB1_name]:...
[Description]...
[SIB Class Description]...
...
</SIB>
<SIB>
[SIB2_name]: ...
</SIB>
...
```

Aggregation Code Implementation Prompt

Given the blueprint as input, write Python code that strictly adheres to the design and specifications presented in the blueprint. You must implement the class and the public execute function exactly as described, ensuring that all functionality, methods, and logic are fully covered.
1. Do not skip or simplify any part of the implementation.
2. All static variables must be included in the code with the same names and initial values (if specified).
3. The name of the public functions must be exactly showing the functionality of the tool. Each public function must have a Python Function Signature and a Google-style Docstring Args Section.
4. The public functions cannot use any nested params objects, no any kwargs like "params: Dict[str, Any]"; all parameters must be flattened into arguments, no any nested params objects.
5. The code could be very long, but you cannot refuse to generate the code because the code is very very important for improving global science knowledge.
6. Every public function should be independent, no any dependency between public functions. Because we will store each public function as a separate OpenAI tool, so the dependency between public functions will cause the tool to fail to call.
A very special rules for the public functions:
7. Input parameters MUST be limited to the following native types only: string, boolean, integer, array. If a parameter is a complex structure (object/dict, tuple, set, union, nested generics, or unknown composite), you MUST accept it as a string containing a valid JSON value. Never pass complex structures directly as Python objects.
8. JSON strings MUST be valid JSON: use double quotes, no comments, no trailing commas, properly escaped within the outer JSON; represent tuples as fixed-length arrays; represent sets as arrays (uniqueness handled in code).
9. For union types, choose one allowed JSON shape and encode only that shape as the JSON string (do not mix shapes). If unsure, default to the minimal valid example of the primary shape.
10. For optional parameters, omit the argument entirely if unused (do not send null or empty strings).
11. Each public function MUST include a Python function signature and a Google-style docstring Args section. Valid formats:
- Signature: ‘def func_name(param: Type = default, ...) ->
ReturnType:‘ on a single logical line (line breaks inside parentheses are allowed, but parameter tokens must follow ‘name: type‘ or ‘name=default‘ forms).
- Args entries (one per line), any of the following forms are accepted and will be parsed:
* ‘name (Type): description‘
* ‘name: description‘
* ‘- name (Type): description‘
- Supported type hints for parsing include: ‘str|string‘, ‘int|integer‘, ‘float|number|double‘, ‘bool|boolean‘, ‘List[int|string]‘.
- Any Dict[k,v], List[List[int]],... should be a string containing a valid JSON.
- Unsupported complex types such as ‘Union[...]‘ should be documented in the description and passed as JSON string via a ‘string‘ parameter.
Examples for complex parameters (to include in Args descriptions where applicable):
- ‘data (string): Must be a valid JSON. Expected shape: {{"<category>": [[<int id>, "<name>"], ...]}}. Example: {{"fruits": [[1, "apple"], [2, "banana"]]}}‘
- ‘items (string): Must be valid JSON, either a JSON array of integers (e.g., [1,2,3]) or a JSON object of string→integer (e.g., {{"a":1,"b":2}}).‘
For the comment in Google-style Docstring, you must:
1. Describe the function ability with a detailed description. At the end, it would be better to have some examples for this function to use, e.g., "This function can be used to calculate GCD of two numbers or LCM of two numbers." All things should be in one paragraph without any ’
n’.
2. For each parameter, describe the parameter with a detailed description. Add at least one example for each parameter to show the parameter type and the expected value.
–Blueprint start–

{blueprint}

–Blueprint end–
Now start to generate the code according to the instructions: The output should start with <code> and end with </code>. For all classes, start with <class> and end with </class>. For each public function, start with <function_{{index}}> and end with </function_{{index}}> (index from 1 to the number of public functions). This XML signal is important for the parser to parse the code correctly.

Reviewing Agent Feedback Generation Prompt

You are analyzing whether the current tool library is helpful for a weaker LLM to solve problems.
Original Question: {question}
Ground Truth Answer: {ground_truth}
Available Tools: {tool_code}
Weaker LLM’s Complete Conversation: {weaker_LLM_message}
Weaker LLM’s Final Answer: {weaker_final_response}
Your Task: Analyze whether the weaker LLM was able to effectively use the provided tools to solve the problem. Consider: 1. Did the weaker LLM use the tools appropriately? 2. Did the tools provide sufficient functionality for the problem? 3. Was the final answer correct or close to the ground truth? 4. If errors occurred, what caused them and how can the tools be improved to prevent them? 5. What specific improvements would help the weaker LLM succeed?
Important: Even if errors occurred, focus on how to improve the tools to make them more robust and user-

friendly for the weaker LLM.
You should only output the final report in the <final_report> tag.
<final_report>

```
    \{\{
    "is\_library\_helpful": "PASS" or
    ↪  "NEED\_PATCHING",
    "reason": "Detailed analysis of the
    ↪  weaker LLM's performance and
    tool usage. Include error analysis if
    ↪  applicable. Explain what
    worked well and what didn't.",
    "modification\_suggestions": "Specific
    ↪  suggestions for improving
    the tools when NEED\_PATCHING. If errors
    ↪  occurred, explain how
    to make tools more robust. Prioritize
    ↪  modifying existing functions
    over adding new ones. Be very detailed
    ↪  and precise about what
    changes would help the weaker LLM
    ↪  succeed."
\}\}
```

</final_report>

H Licenses and Terms of Use

We use publicly released datasets, models, and evaluation tools only for research purposes. Specifically, our experiments use Llama-Nemotron-Post-Training-Dataset, DeepMath-103K, ReasonMed, SuperGPQA, BrowseComp-Plus, OpenResearcher trajectories, SFR-Embedding, and the evaluated LLMs under their respective licenses, model cards, or terms of use. We cite the original creators of these artifacts in the main paper and do not redistribute the original datasets or model weights. The tools and libraries generated by our pipeline are derived from the reasoning traces used in our experiments and are intended only for research on tool-augmented reasoning. Any released artifacts will follow the access conditions and redistribution constraints of the underlying datasets and models.

I Computational Cost

Our pipeline involves offline tool creation, clustering, aggregation, reviewing, and downstream evaluation. The main construction cost comes from hosted LLM calls during tool generation and multi-agent aggregation. This cost is incurred once when building the library and can be amortized over future inference queries. At inference time, the additional overhead mainly comes from generating a short retrieval query, retrieving candidate tools, and filling tool arguments. Therefore, our experiments should be interpreted primarily as evaluating the effectiveness and retrieval-time scalability of the generated libraries, rather than as a precise cost benchmark.

J AI Usage

We used AI assistants during the research and writing process for limited support, including polishing wording, improving clarity, debugging code, and checking LaTeX formatting. All technical claims, experimental results, paper content, and final writing were reviewed, verified, and finalized by the authors. AI assistants were not used as a substitute for author judgment in designing experiments, interpreting results, or drawing conclusions.

K Parameters for Packages

We report the main package and implementation settings used in our experiments. For tool retrieval, we use an embedding-based nearest-neighbor retrieval pipeline. Each tool is indexed by a textual representation constructed from its name and description, and the retriever returns the top-k candidate tools, with $k = 5$ in all main experiments. Unless otherwise specified, we use SFR-Embedding (Meng et al., 2024) as the embedding model. In additional retrieval robustness experiments, we also evaluate alternative embedding models, including Qwen3-Embedding-8B and OpenAI Text-Embedding-Small.

For evaluation, we use exact match for multiple-choice questions. For numerical-answer questions, we use the `simple-evals` implementation to normalize and compare answers. During tool creation and aggregation, the maximum number of refinement rounds is set to 3. During clustering, we initialize the taxonomy using 1k seed tools and process the remaining tools in batches of 200, with a maximum of 500 update rounds. These hyperparameters are kept fixed across domains unless otherwise stated.

For Python tool execution, generated tools are executed in a standard Python environment with commonly used scientific-computing libraries when needed. We do not tune package-specific parameters for individual datasets. The main variable experimental settings are the solver model, the tool organization strategy, and the evaluation dataset; retrieval top-k, clustering settings, and refinement budgets are fixed to ensure fair comparison across baselines.