Toolink: Linking Toolkit Creation and Using through Chain-of-Solving on Open-Source Model

Cheng Qian[1], **Chenyan Xiong**[2], **Zhenghao Liu**[3], **Zhiyuan Liu**[1]
[1]Tsinghua University, [2]Carnegie Mellon University, [3]Northeastern University
qianc20@mails.tsinghua.edu.cn

Abstract

Large Language Models (LLMs) have demonstrated remarkable progress in utilizing tools, but their closed-source nature and high inference costs pose limitations on their adaptability, necessitating a valid method that leverages smaller, open-sourced models. In this paper, we introduce Toolink, a comprehensive framework that performs task-solving by first creating a toolkit and then integrating the planning and calling of tools through a chain-of-solving (CoS) approach. We first validate the efficacy of Toolink in harnessing the model's creativity and CoS ability on ChatGPT. Subsequently, we curate CoS-GPT, a chain-of-solving dataset designed for tool-using, and finetune the LLaMA-7B model. It results in LLaMA-CoS, a powerful open-source model with advanced tool-planning and tool-calling capabilities. Evaluation on diverse tasks from BIG-bench demonstrates its CoS ability matches that of ChatGPT while its performance surpasses the chain-of-thought approach. Further studies highlight the generalization of LLaMA-CoS to unseen tasks and showcase its capability in using toolkits not explicitly tailored for the target task, affirming its robustness in real-world scenarios. All codes and data are released[1].

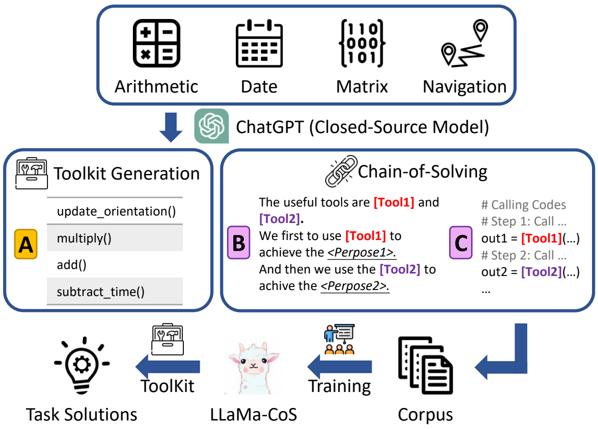


Figure 1: An illustration of Toolink. It decomposes the tasks through toolkit creation, and solves the queries through chain-of-solving (CoS). Toolink can be adapted on open-source LLaMA for effective tool-using.

1 Introduction

Large Language Models (LLMs) such as Codex (Chen et al., 2021), ChatGPT (OpenAI, 2022), and GPT4 (OpenAI, 2023) have made significant advancements in code generation, in-context learning, and logical reasoning. However, these models still face limitations in precise calculations and accessing up-to-date information (Patel et al., 2021; Trivedi et al., 2022; Lu et al., 2022b). To overcome these challenges, recent research has focused on equipping LLMs with tools to enhance their expertise and interpretability (Qin et al., 2023). These tools, such as calculators (Cobbe et al., 2021; Parisi et al., 2022; Schick et al., 2023), search engines (Carlini et al., 2021; Thoppilan et al., 2022; Schick et al., 2023), scratch pads (Nye et al., 2021), calendars (Schick et al., 2023), and image retrievers (Sheynin et al., 2022), empower LLMs to access external resources, benefiting various tasks including question-answering, math calculations, and long-form generation. Recent studies have also conducted some attempts to leverage LLMs to plan and utilize these tools (Shen et al., 2023; Lu et al., 2023; Liang et al., 2023). By combining plans, decisions, and executions into a pipeline, these frameworks aim to construct more advanced and versatile NLP systems for improving the performance of LLMs.

However, current tool-using pipelines heavily rely on closed-source models with inaccessible parameters. It poses challenges particularly in the following aspects: (1) **Limited adaptability**: The closed-source nature of LLMs prevents them from being easily finetuned, resulting in a lack of flexibility to adapt to customized tasks according to specific requirements. (2) **Low efficiency and high**

[1]https://github.com/qiancheng0/Toolink

inference cost: Many existing LLMs can only be accessed *online*, which imposes limitations on the inference rate and leads to high costs. (3) **Privacy and security concerns**: Each query must be submitted to these closed-source LLMs to obtain a tool-using solution, which raises legitimate concerns regarding potential privacy breaches and compromises in data security.

To address these challenges, we propose Toolink, a comprehensive framework to boost the tool-using ability of open-source LLMs with the help of the tool-using capability of ChatGPT. As shown in Figure 1, it first decomposes the target task by creating a toolkit for problem-solving, and then leverages the model to use tools to answer queries in a chain-of-solving (CoS) approach. Specifically, CoS is further disentangled into two distinct aspects: *CoS-Planning*, which selects useful tools from the created toolkit and plans their usages based on the specific query; and *CoS-Calling*, which focuses on deriving the answer by calling the tools in code format according to plans. Finally, we employ ChatGPT to curate CoS-GPT, a training dataset that aims to inspire the open-source model's tool-using capability through CoS. Specifically, we finetune LLaMA-7B (Touvron et al., 2023) into LLaMA-CoS, which is enabled with tool-using capabilities by linking the created toolkit with the chain of problem-solving.

LLaMA-CoS can solve the queries *offline* without uploading queries to closed-source models, ensuring data security and privacy. Experiments illustrate that Toolink outperforms the chain-of-thought (CoT) (Wei et al., 2022) on diverse tasks from BIG-bench (Srivastava et al., 2022) and enables LLaMA-CoS to showcase comparable CoS ability to that of ChatGPT. In addition, LLaMA-CoS can generalize to unseen tasks by planning and calling tailored tools, and solve the target task with a toolkit not specifically tailored for it. These findings further affirm our framework's robustness in solving real-world queries.

2 Related Work

Tool-based enhancement for LLMs. Language models have been enhanced with external tools to improve their expertise. Previous work focused on equipping the LLMs with different tools including calculator to improve calculation accuracy (Cobbe et al., 2021; Parisi et al., 2022; Schick et al., 2023), search engine to inquire factual knowledge (Carlini et al., 2021; Thoppilan et al., 2022; Schick et al., 2023), Python interpreter to execute programs (Chen et al., 2022a; Gao et al., 2022), and retriever to search textual information (Khandelwal et al.; Borgeaud et al., 2022), etc.

More recent studies, such as HuggingGPT (Shen et al., 2023), Chameleon-LLM (Lu et al., 2023), VisualGPT (Wu et al., 2023) and TaskMatrix.AI (Liang et al., 2023), focus on assembling plannings, execution, and logical reasoning on tools into a robust pipeline. In addition to tool-using, ART (Paranjape et al., 2023) builds toolkits based on retrieved tasks from the manually built library, while LATM (Cai et al., 2023) and CREATOR (Qian et al., 2023) involves the LLMs' tool-making ability to offload their reasoning burden and raise task performance.

Adaptation of open-source models. One research direction focuses on effective tuning of open-source models, including the introduction of lightweight modules such as Adapter (Houlsby et al., 2019) and LoRA (Hu et al., 2021). These modules are adapted to various model types including LLaMA (Touvron et al., 2023), T5 (Raffel et al., 2020), and other Transformers-based architectures (Pfeiffer et al., 2020), to save computational resources and improve effectiveness. For instance, GOAT (Liu and Low, 2023) applies LoRA to improve LLaMA's arithmetic calculation ability, while LLaMA-Adapter (Zhang et al., 2023) adopts Adapter and zero-init attention to improve multi-modal task performance.

Other works have investigated how instruction tuning can make open-source models better understand and follow the instructions. Among these, Flan (Longpre et al., 2023) explores the methods of instruction tuning while InstructGPT (Ouyang et al., 2022) further improves its effectiveness with human feedback. More recent works also extend instruction tuning to visual domains (Liu et al., 2023) and leverage the LLMs to build the instruction-following data to improve open-source models (Taori et al., 2023; Peng et al., 2023).

3 Method

As shown in Figure 2, Toolink first adopts toolkit creation to break down the target task through generating potential tools for task-solving (Sec. 3.1). Then, the model links these created tools to address specific queries by selecting pertinent tools from the toolkit, planning their uses, and making

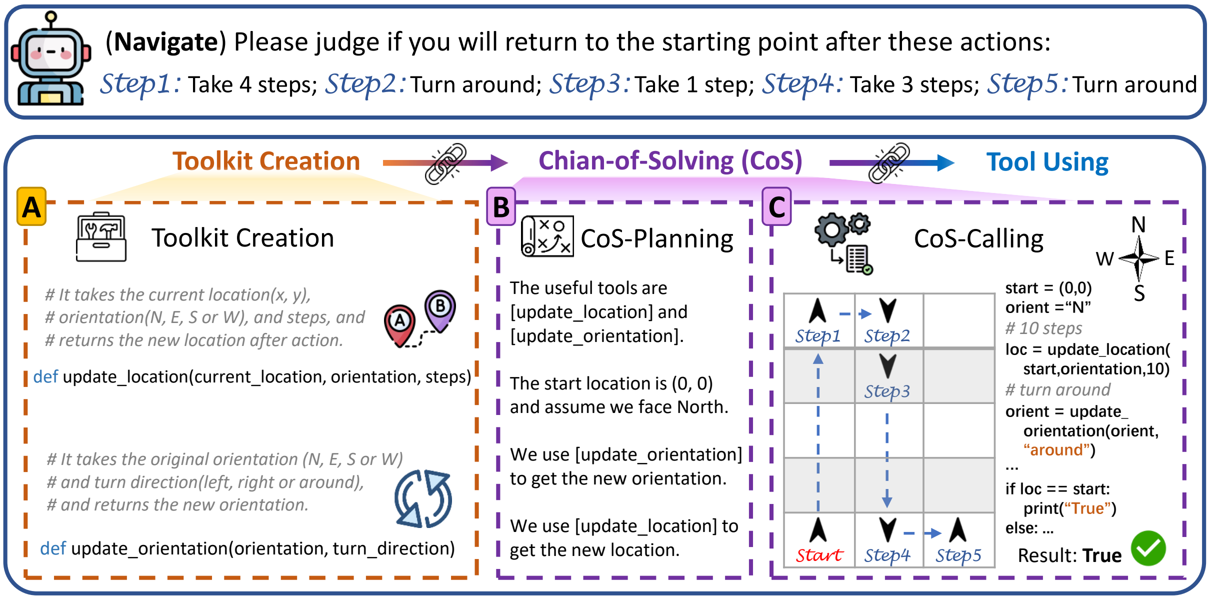


Figure 2: A problem solving chain of Toolink pipeline. We show an example from task Navigate. Toolink first creates a toolkit generally applicable to the task, and then approaches the specific query through CoS, which involves planning and calling of the created tools.

the final calling decision (Sec. 3.2). This process, referred to as chain-of-solving (CoS), not only enables the effective and coherent application of tools but also facilitates the tool-using adaptation on the open-source model (Sec. 3.3).

3.1 Toolkit Creation

Given the target task T, toolkit creation aims to break down T into more manageable components $t_1, t_2, ..., t_n$ through generating a toolkit $K_T = \{k_1, k_2, ..., k_n\}$ where $k_i (i \leq n)$ represents the tool to solve a subtask t_i. This step tries to decompose a general task into modular and essential tools for problem-solving, which facilitates more flexible tool utilization.

Toolkit Making. For the task T, we utilize ChatGPT to decompose the task into different tools. We feed the ChatGPT model with task meta information I_T and s publicly available target task samples $D_{T\text{-sample}} \subset D_T$ to create the toolkit K_T:

$$\text{Instruction} + I_T + D_{T\text{-sample}} \xrightarrow{\text{ChatGPT}} K_T, \quad (1)$$

where $D_{T\text{-sample}} \subset D_T$ and D_T consists of all data points for the task T. Specifically, we first involve clear instructions to demonstrate the toolkit creation task for ChatGPT. Then we give the task meta information I_T and some instances $D_{T\text{-sample}}$ to expect ChatGPT to better understand the objective of the given task T and identify commonalities among queries, thereby successfully decomposing T into more manageable units. More details are showcased in Appendices A.

Note that our design only necessitates a limited amount of s data points to be fed into the closed-source ChatGPT, while leaving $D_{T\text{-remain}} = D_T \setminus D_{T\text{-sample}}$ to be processed locally to ensure privacy is maintained ($|D_{T\text{-remain}}| \gg |D_{T\text{-sample}}|$).

Tool Details. Each tool k_i within the toolkit K_T is comprised of a concise introduction and its corresponding code implementation. The introduction provides a brief overview of k_i's utility, inputs, and outputs, facilitating effective planning and calling in subsequent steps. In Figure 2A, the target task $T = \texttt{Navigate}$ is decomposed into t_1 (movement in a single direction) and t_2 (change of orientation). Each component is represented by a specific implementation encapsulated within a tool $k_i (i \in \{1, 2\})$, which offers increased flexibility when applying CoS for specific problem-solving.

3.2 Chain-of-Solving

Chain-of-Solving (CoS) involves dynamically selecting useful tools K_{use} from the created toolkit K_T for each query Q and planning their uses with calling decisions. It links the potentially useful tools in the created toolkit and is disentangled into CoS-Planning and CoS-Calling, which facilitates a more transparent and interpretable reasoning path

Category	Set Name	Source	Number
Tool-Using	Tool-Planning	Augmented	4.4K
	Tool-Calling	Augmented	4.4K
Code Generation	Python-Simple	New	2.0K
	Python-Specific	New	2.0K
	Math	Augmented	2.5K
	Algorithm	Github	2.3K
	LeetCode	LeetCode	0.8K
	Rectification	Sources Above	1.6K
Total	-	-	20.0K

Table 1: The statistics about the sources and number of data points in each category of CoS-GPT. *Augmented* represents augmented from an existing dataset.

and enhances its applicability to open-source models.

CoS-Planing. Planning in CoS involves intelligently selecting useful tools K_{use} from a given toolkit K_T, and utilizing natural language based reasoning chains (Plan) to determine how to employ K_{use} to solve a specific query $Q \in T$:

$$K_T + Q \xrightarrow{\text{ChatGPT}} K_{\text{use}} + \text{Plan}. \tag{2}$$

In Figure 2B, the model devises strategies for employing tools to update the location and orientation, with additional initial conditions that may serve as a guiding hint later. Planning plays a crucial role in establishing a link between toolkit creation and decision-making, thus reducing the cognitive burden associated with tool-use reasoning.

CoS-Calling. Calling entails utilizing K_{use} and interpreting the tool-using plans by regarding the program language as a bridge. Plannings in the previous step serve as guidance to generate the program implements $\texttt{Impl}\{K_{\text{use}}\}$. During execution, all tool calling results will be implicitly captured to generate the final answer A for query Q:

$$Q + K_{\text{use}} + \text{Plan} \xrightarrow{\text{ChatGPT}} \texttt{Impl}\{K_{\text{use}}\} \xrightarrow{\text{Exec}} A. \tag{3}$$

In Figure 2C, the model simulates the whole navigation process leveraging code and derives the ultimate correct answer, thereby exemplifying a successful calling decision.

3.3 Open-Source Model Adaptation

The Toolink framework introduced previously mainly stimulates a closed-source model, ChatGPT, to create and use tools. However, considering the limited adaptability, high inference cost, and privacy concerns, we aim to transfer the CoS ability of ChatGPT to open-source model M_{open} by tuning it properly. To this end, we introduce CoS-GPT, a training dataset focusing on the planning and calling of tools as well as code generation, all of which serve as the fundamentals in promoting M_{open}'s CoS ability. We denote CoS-GPT as D_{CoS} in the following and present its statistics in Table 1. Additionally, for each target task T, we utilize $D_{T\texttt{-sample}}$ to create a task-specific dataset $D_{T\texttt{-tool}}$, which augments each query with tools and enables more effective training of task T on open-source models, M_{open}.

Construction of CoS-GPT. To enhance M_{open}'s skills in applying tools for problem-solving, we construct D_{CoS} from scratch to improve its CoS ability from planning, calling, and coding. We include the first two aspects as they are essential for CoS within Toolink, and the last aspect as it serves as the medium for tool-using.

For data points about planning and calling, we enhance the existing AQUA-RAT (Ling et al., 2017), GSM8K (Cobbe et al., 2021), and TabMWP (Lu et al., 2022a) datasets by incorporating tools. These datasets consist of graduate-level math problems, numerical reasoning tasks, and diverse table contents respectively. We augment each query with a toolkit, which contains both the useful and redundant tools for this specific query. For planning, we aim to let M_{open} select useful tools K_{use} from the toolkit and plan their uses. For calling, we aim to let M_{open} learn how to call K_{use} in codes to solve the problem. During data construction, we apply ChatGPT to simulate this process and utilize their responses to construct the dataset. Please refer to Appendices E.1 for more details.

The data construction for code generation and understanding encompasses diverse sources, including augmentation from existing datasets, GitHub repositories, and newly generated data, detailed in Appendices E.2. Each query adheres to an instruction-following pattern and aims to enhance M_{open}'s understanding of code while expanding its versatility in making informed decisions when performing CoS.

Target Task-Specific Data. Suppose we have a set of target tasks $T_{\texttt{all}}$. For each $T_i \in T_{\texttt{all}}$, we construct 200 tool-augmented data points $D_{T_i\texttt{-tool}}$ (100 each for plan and call) from the publicly available samples $D_{T_i\texttt{-sample}}$, and use them to tune

M_{open} together with D_{CoS}. Similar to the construction process for tool-using data in D_{CoS}, we first augment T_i with a toolkit K_{T_i}. Next, we employ ChatGPT to select useful tools for each query and generate the calling decision. The decision's output is compared against the standard answer, and minor adjustments may be made to ensure the validity of these newly-constructed tool-augmented data.

Finetuning of Model. Together with D_{CoS}, we apply the tool-augmented samples $D_{T_i\text{-tool}}$ from all target tasks to finetune M_{open}:

$$M_{\text{open}} \xrightarrow{D_{\text{CoS}} \cup \{D_{T_i\text{-tool}} | T_i \in T_{\text{all}}\}} M_{\text{tool}}. \quad (4)$$

We expect the derived tool-augmented open-source model M_{tool} to have the CoS ability to apply useful tools in the toolkit for problem-solving. With excellency in planning and calling, M_{tool} links the created toolkit with specific queries, which realizes the final goal of the Toolink framework.

4 Experiments

To evaluate the effectiveness of Toolink, we initially conduct a validation test utilizing the ChatGPT model. We select eight distinct tasks from the BIG-bench dataset (Srivastava et al., 2022) to investigate whether Toolink can effectively leverage ChatGPT's creativity and tool-using capability to improve task performance.

Subsequently, we perform finetuning on the open-source LLaMA-7B model by following the adaptation process outlined in Section 3.3. This results in LLaMA-CoS, which links the created toolkit with specific tool use through CoS. We then assess the effectiveness of LLaMA-CoS in utilizing tools on the same set of eight tasks and showcase its excellence.

4.1 Validation Evaluation

Settings. In order to assess the effectiveness of Toolink, we conducted a validation test utilizing ChatGPT and select eight distinct tasks from BIG-bench. The tasks include Arithmetic, Date Understanding, Matrix Shape, Navigate, Chinese Remainder, Dyck Language, Boolean Expression, and Tracking Shuffled Objects.

For each task, we initially employ ChatGPT in the creation of a toolkit, outlined in Section 3.1. We statistic the total number of tools in the toolkit for each task and showcase it in Table 2. Appendices B provide the specific tools for each task. Equipped with these tools, the model is presented with instructions and demonstration examples in the chain-of-solving stage to guide it link tools for problem-solving, detailed in Appendices C.

Baselines. We compare our approach against two baselines: the **Vanilla** baseline, where ChatGPT directly produces the final answer, and the **CoT** baseline (Wei et al., 2022), where ChatGPT employs a chain-of-thought approach to produce the reason chain for the query before providing an answer.

Evaluation Methods. We explore the ability of ChatGPT in leveraging plans and calls together into a pipeline to perform CoS. The accuracy is measured by matching the ChatGPT's final output to the correct answer.

To comprehensively analyze the individual contributions of CoS-Planning and CoS-Calling, we also evaluated their accuracy separately. For CoS-Planning, the model is asked to only select useful tools and plan their utilization given the query and the created toolkit, as outlined in Formula 2. The accuracy is measured by the following metric:

$$ACC = \max\{\frac{|K_{\text{correct}}| - |K_{\text{error}}|}{|K_{\text{correct}}| + |K_{\text{error}}|}, 0\}, \quad (5)$$

where $|K_{\text{correct}}|$ denotes the number of correct (useful) tools in the toolkit selected in the model's generated plan, while $|K_{\text{error}}|$ denotes the number of erroneous (redundant) tools selected.

For CoS-Calling, the model is asked to implement the plan using code as the medium, given the query and useful tools in the toolkit, as outlined in Formula 3. The accuracy is measured by matching the output from the final execution with the correct answer. For more details regarding the separation of CoS-Planning and CoS-Calling tests, please refer to Appendices D.

Results. The results are presented in Table 2. ChatGPT that utilizes tools through the CoS approach achieves significantly improved performance compared to other baselines, with notable margins of superiority. Further, the accuracy for CoS-Calling and CoS-Planning individually is even higher, indicating successful reasoning in each step of CoS which links toolkit creation with specific uses. These findings affirm the validity of Toolink, establishing a strong basis for its potential transferability to smaller, open-sourced models.

Task	Arith.	Date U.	Matrix S.	Navigate	Chinese R.	Dyck L.	Boolean E.	Tracking S.	Average
Num. of Tools	5	3	5	2	2	4	2	4	3.38
Vanilla	77.78	68.67	40.90	65.16	0.0	19.40	80.70	23.67	47.03
CoT	79.44	68.67	80.46	87.96	0.0	19.42	75.88	40.78	56.58
CoS	100.00	69.28	93.67	85.30	95.14	52.46	97.37	99.11	**86.54**
CoS-Planning	100.00	66.16	95.18	94.78	100.00	74.58	95.39	99.85	90.74
CoS-Calling	100.00	90.96	97.44	88.44	95.67	98.55	93.42	100.00	95.56

Table 2: We first record the number of tools in the toolkit created for each task. Next, we demonstrate the accuracy (%) of ChatGPT under different settings on 8 tasks sourced from BIG-bench. We report the results of Vanilla, CoT baselines and our CoS method. We also report the performance of CoS-Planning and CoS-Calling separately.

Method	Model	Arith.	Date U.	Matrix S.	Navigate	Chinese R.	Dyck L.	Boolean E.	Tracking S.
CoT	Alpaca	19.89	39.76	5.62	47.11	0.0	0.0	57.46	0.44
(*Zero-shot*,	LLaMA-7B	39.44	33.73	12.58	39.70	0.0	2.90	50.44	14.22
w/ demo)	ChatGPT	79.44	68.67	80.46	87.96	0.0	19.42	75.88	40.78
CoT (*Tuned*)	LLaMA-CoT	50.44	49.40	70.82	71.64	0.0	35.27	62.72	28.44
CoS	Alpaca	17.78	7.83	3.00	48.60	7.56	1.00	94.74	6.78
(*Zero-shot*,	LLaMA-7B	55.89	17.47	10.65	45.90	23.80	35.83	99.12	0.67
w/ demo)	ChatGPT	**100.00**	69.28	**93.67**	85.30	95.14	52.46	97.37	99.11
CoS (*Tuned*)	LLaMA-CoS	**100.00**	**74.10**	91.01	**99.56**	**95.44**	**98.21**	**100.00**	**99.56**

Table 3: The accuracy (%) on the 8 tasks sourced from BIG-bench. We report the baseline results from three models including LLaMA-7B, Alpaca, and ChatGPT. LLaMA-CoS employs planning and calling of tools, which beats all CoT baselines by large margins and is on par with ChatGPT's CoS ability.

4.2 Experiments on LLaMA-CoS

Considering the limitations associated with the closed-source models, our primary objective is to extend Toolink to smaller, open-sourced models. Among these, the models from LLaMA family (Touvron et al., 2023) stand out due to their capability to perform reasoning, follow in-context examples, and generate codes. Considering the affordability of computational resources, we select LLaMA-7B as the representative base model to evaluate the performance of Toolink on open-source models.

Obtaining LLaMA-CoS. We follow the adaptation process outlined in Section 3.3 and finetune LLaMA-7B with the CoS-GPT we introduced ($D_{\texttt{CoS}}$) and eight sets of task-specific tool-augmented data ($D_{T_i\text{-}\texttt{tool}}, 1 \leq i \leq 8$). The eight target tasks are the same ones we apply in Section 4.1. Through the training detailed in Appendices F, we derive a powerful variant, LLaMA-CoS, that excels in using tools through CoS.

Settings. We utilize LLaMA-CoS as the representative finetuned open-source model. Building upon the validation test conducted on ChatGPT, we further evaluate its performance on the same set of eight tasks obtained from BIG-bench.

Baselines. As a comparison to CoS, we employ the chain-of-thought (CoT) reasoning as the baseline. We evaluate both methods under two scenarios: (1) zero-shot prompting with demonstrations on Alpaca, LLaMA-7B, and ChatGPT, and (2) normal finetuning specifically on the LLaMA-7B model. We referred to the LLaMA-7B tuned with CoT data as LLaMA-CoT, while our LLaMA-CoS is tuned with data points specially designed to enhance its ability to use the tools.

Results. We present the results in Table 3. Notably, LLaMA-CoS achieves an impressive average accuracy of 94.74%, outperforming all the CoT baselines, whether tuned or not, by a substantial margin. Even compared to ChatGPT, which exhibits strong reasoning and tool-using capabilities under the CoS setting, our tuned model can still achieve comparable performance. These results highlight the effectiveness of CoS in outperforming traditional CoT methods and demonstrate the successful transfer of tool-using abilities from closed-

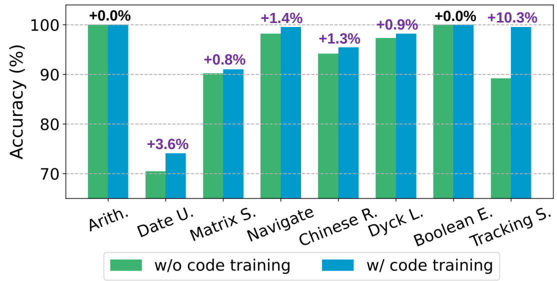


Figure 3: The improvement of performance when code generation data points are involved for each task.

source LLMs to smaller, open-source models.

4.3 Results Analysis

Excellence in Both Planning and Calling. To assess the effectiveness of both planning and calling during CoS, we conduct additional studies specifically targeting these two aspects in Table 4. We separate these two steps in the same way as we described in the evaluation methods of Section 4.1. Remarkably, our experimental results demonstrate CoS-Planning and CoS-Calling separately surpass the performance achieved by CoT-based models on all tasks. Moreover, the accuracy of the complete CoS pipeline is approximately equivalent to the product of the accuracy of CoS-Planning and CoS-Calling. These findings serve to validate the model's proficiency in performing well on each individual step. Furthermore, they underscore the critical role played by planning and calling in ensuring the success of the whole CoS reasoning, thereby providing evidence supporting the rationale behind the development of the planning and calling steps during CoS for effective tool-using under Toolink framework.

Necessity of Code Training. To evaluate the impact of code generation data, we compare the effectiveness of finetuned LLaMA-7B on the tasks with and without code generation data points in the base training set. The results, presented in Figure 3, clearly indicate that our LLaMA-CoS trained with code generation data achieves significantly higher accuracy. On average, the inclusion of code generation data leads to a performance improvement of approximately 1.4%. These findings provide strong evidence supporting the necessity of integrating code generation training when learning tools and CoS ability. By incorporating code generation data, the model effectively learns to utilize code as a medium for tool-using, which helps them adapt to different scenarios with more flexibility and ultimately results in enhanced performance.

Diverse Usage of Toolkit. We discover throughout the experiments that LLaMA-CoS exhibits diverse CoS-planning and CoS-calling patterns for tool-using. It is capable of sequentially calling different tools to achieve a specific purpose, using tools based on the condition given through a non-linear logic, or performing nested tool calls, where the output from one tool directly serves as the other one's input. These abilities illustrate the robustness and adaptability of LLaMA-CoS across diverse scenarios. We provide three case studies and more details in Appendices G and Figure 16.

5 Further Studies

In this section, we show the generalization of LLaMA-CoS to novel tasks and how it can apply CoS in using tools that are not specially tailored for solving the target task. These studies aim to show the robustness of LLaMA-CoS in utilizing tools through planning and calling.

5.1 Generalization to Novel Tasks

The eight evaluation tasks (Srivastava et al., 2022) we used in the previous experiment have all been presented in the training data, even though we only leverage a few tool-augmented publicly available samples. To showcase the generalization ability of LLaMA-CoS, we further test it on two new tasks: FinQA (Chen et al., 2022b) and GSM8K (Cobbe et al., 2021). FinQA involves question-and-answer pairs based on financial report data, while GSM8K focuses on grade school math problems.

Together with AQUA-RAT, MATH, and TabMWP, whose data are presented in CoS-GPT (detailed in Section 3.3), we randomly select a maximum of 400 test data points from each of the five tasks, and ensure they do not appear in CoS-GPT. We augment each data point with a toolkit, following the method outlined in Section 3.3 regarding how CoS-GPT is constructed. For the experiment, we follow the CoS-planning and CoS-calling test process outlined in the evaluation methods of Section 4.1.

Table 5 presents the statistics and testing results of LLaMA-CoS. We show that it achieves high accuracy in both the tool-calling and tool-planning steps, affirming the effectiveness and robustness of its CoS ability for tool-using even when applied to unseen tasks. These findings also emphasize **the**

Method	Model	Arith.	Date U.	Matrix S.	Navigate	Chinese R.	Dyck L.	Boolean E.	Tracking S.
CoS-Whole	LLaMA-CoS	**100.00**	74.10	91.01	99.56	95.44	98.21	100.00	99.56
CoS-Planning	Alpaca	18.22	27.41	24.15	77.16	**100.00**	76.3	**97.59**	99.37
(*Zero-shot,*	LLaMA-7B	74.11	27.71	25.02	77.16	**100.00**	93.80	**97.59**	**100.00**
w/ demo)	ChatGPT	**100.00**	66.16	**95.18**	94.78	**100.00**	74.58	95.39	99.85
CoS-Planning	LLaMA-CoS	**100.00**	**85.84**	89.62	**97.14**	**100.00**	**99.19**	**97.59**	**100.00**
CoS-Calling	Alpaca	99.44	24.70	30.08	48.60	17.97	1.56	89.91	6.78
(*Zero-shot,*	LLaMA-7B	74.70	51.20	55.49	43.77	24.81	25.67	94.30	1.56
w/ demo)	ChatGPT	**100.00**	90.96	**97.44**	88.44	**95.67**	**98.55**	93.42	**100.00**
CoS-Calling	LLaMA-CoS	**100.00**	**91.57**	95.56	**98.88**	94.18	**98.55**	**95.61**	88.44

Table 4: The accuracy (%) of CoS-Planning and CoS-Calling separately on 8 tasks sourced from BIG-bench. Results show LLaMA-CoS has excellent ability in understanding and using tools through CoS.

Task	In D_{CoS}	Count	CoS-Planning	CoS-Calling
AQUA-RAT	✓	139	59.80	56.12
MATH	✓	400	65.83	50.75
TabMWP	✓	400	90.00	66.00
FinQA	✗	210	70.51	22.38
GSM8K	✗	400	61.29	57.25

Table 5: The accuracy (%) of CoS-Planning and CoS-Calling on five diverse datasets applying LLaMA-CoS. D_{CoS} represents the CoS-GPT dataset. Results show LLaMA-CoS is robust to unseen tasks *w.r.t.* tool-using.

Task	Toolkit Origin	LLaMA-CoS	ChatGPT
Dynamic Cnt.	*Raw*	97.50	80.83
	From Dyck L.	73.30	79.17
Unit Interp.	*Raw*	70.83	80.83
	From Arith.	65.83	80.00

Table 6: The accuracy (%) of ChatGPT and LLaMA-CoS, with toolkit newly created for the target task (*Raw*) or borrowed from other tasks. Our results show that both ChatGPT and LLaMA-CoS can utilize tools not specifically tailored for the target task through CoS.

generalization capabilities and the robustness of LLaMA-CoS across diverse domains, showing its wide applicability.

5.2 CoS on Generic Toolkit

We further explore the ability of LLaMA-CoS to use generic toolkits instead of the one specifically tailored for the target task. In real-world scenarios, toolkits are usually designed to address tasks across diverse domains, rather than tailored specifically for a single task. We assume that LLaMA-CoS and ChatGPT can also apply toolkits borrowed from other tasks to solve target queries in a CoS approach and achieve comparable performance.

To validate our assumption, we source two additional tasks from BIG-bench: Dynamic Counting and Unit Interpretation. For each task, we provide a toolkit that is either created explicitly for the target task or borrowed from another task. Specifically, we pair Dynamic Counting and Unit Interpretation with Dyck Language and Arithmetic, respectively.

We evaluate the performance of each setting using LLaMA-CoS and ChatGPT. The results presented in Table 6 indicate that **both LLaMA-CoS and ChatGPT can utilize a generic toolkit borrowed from another task to solve target queries through CoS**. Though the performance still lags behind using the toolkit specifically tailored for the target task, these findings nevertheless confirm our assumption that CoS has the ability to help increase the robustness of tool-using, and make our framework more applicable to real-world scenarios. More details are shown in Appendices H.

6 Conclusions

We present Toolink, a tool-training framework that effectively applies toolkits to solve problems leveraging small, open-source language models. Toolink offers increased flexibility in adapting to diverse downstream tasks while addressing concerns related to high inference costs and privacy. Our main contributions include (1) empirically implementing a framework that can effectively leverage open-source models' tool-using ability, (2) devising the chain-of-solving (CoS) method that links toolkit creation and uses through robust planning and calling, and (3) releasing the CoS-GPT dataset that enhances the model's CoS capabilities.

Specifically, our LLaMA-CoS outperforms traditional CoT and achieves a comparable performance to ChatGPT with respect to tool-using. We believe our study provides a solid foundation and serves as inspiration for future researchers to further explore the potential of enhancing open-source models with advanced tool-using capabilities.

Limitations

Our experiments focus on equipping the open-source model with tool-using capabilities through the CoS approach, specifically in planning and calling, while excluding the ability to create toolkits. This limitation arises from the fact that the LLaMA-7B primarily relies on provided demonstrations and lacks the internal creativity required for toolkit creation. Moreover, the absence of enough training data further hampers the acquisition of this knowledge. We acknowledge this challenge posed by the transfer of the toolkit creation capability from closed-source LLMs and leave it as an avenue for future research.

Additionally, it is important to note that though the tasks tested in our study include diverse toolkits and queries, they are mostly drawn from the BIG-bench dataset. To gain a more holistic understanding of the generalizability of our results, it is imperative for future research to expand the application of Toolink to a broader range of scenarios. This expansion would enable a more comprehensive assessment of the framework's efficacy and applicability across diverse domains.

Ethics Statement

We consider the following issues in this paper:

- **Privacy** is a crucial aspect to consider when utilizing closed-source models such as ChatGPT and GPT4. These models have the potential to learn sensitive information internally, posing a risk to personal privacy. In contrast, Toolink addresses this concern by leveraging only a limited number of publicly available samples for toolkit creation, leaving the majority of testing queries blind to closed-source LLMs. This approach reduces the possibility of mishandling data and safeguards user privacy. By minimizing the exposure of sensitive information, Toolink mitigates the risks associated with privacy breaches when compared to closed-source models.
- **Transparency** is a key aspect that aims to enhance the interpretability and comprehensibility of AI systems from a human perspective. In our framework, we prioritize transparency through the creation of toolkits that provide clear information about their utility, inputs, and outputs. Additionally, we disentangle the CoS into separate steps of planning and calling, which increases the interpretability of the model's reasoning for users. We also encourage future research to further document the specific scenarios in which our framework exhibits its maximum effectiveness, as well as to outline potential risks involved. This will contribute to a more comprehensive understanding of our framework and facilitate informed decision-making.
- **Potential Bias** is another critical aspect that we prioritize addressing in our work. We acknowledge that bias and discrimination can inadvertently manifest through problematic examples present in the training data. To mitigate this concern, we adopt a meticulous approach to curate the CoS-GPT dataset, which consists of data points from various sources. We emphasize diversity to minimize the presence of potentially biased patterns during the data construction. Through these efforts, we aim to develop the model's tool-using and CoS ability that promotes equitable and unbiased outcomes, fostering trust and inclusiveness in the application of AI systems.

References

Aida Amini, Saadia Gabriel, Peter Lin, Rik Koncel-Kedziorski, Yejin Choi, and Hannaneh Hajishirzi. 2019. Mathqa: Towards interpretable math word problem solving with operation-based formalisms.

Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George Bm Van Den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, et al. 2022. Improving language models by retrieving from trillions of tokens. In *International conference on machine learning*, pages 2206–2240. PMLR.

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. 2023. Large language models as tool makers.

Nicholas Carlini, Florian Tramer, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, Ulfar Erlingsson, Alina Oprea, and Colin Raffel. 2021. Extracting training data from large language models.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

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

Zhiyu Chen, Wenhu Chen, Charese Smiley, Sameena Shah, Iana Borova, Dylan Langdon, Reema Moussa, Matt Beane, Ting-Hao Huang, Bryan Routledge, and William Yang Wang. 2022b. Finqa: A dataset of numerical reasoning over financial data.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2022. Pal: Program-aided language models. *arXiv preprint arXiv:2211.10435*.

Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-efficient transfer learning for nlp. In *International Conference on Machine Learning*, pages 2790–2799. PMLR.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models.

Urvashi Khandelwal, Omer Levy, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. Generalization through memorization: Nearest neighbor language models. In *International Conference on Learning Representations*.

Yaobo Liang, Chenfei Wu, Ting Song, Wenshan Wu, Yan Xia, Yu Liu, Yang Ou, Shuai Lu, Lei Ji, Shaoguang Mao, et al. 2023. Taskmatrix. ai: Completing tasks by connecting foundation models with millions of apis. *arXiv preprint arXiv:2303.16434*.

Wang Ling, Dani Yogatama, Chris Dyer, and Phil Blunsom. 2017. Program induction by rationale generation: Learning to solve and explain algebraic word problems. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 158–167.

Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. 2023. Visual instruction tuning. *arXiv preprint arXiv:2304.08485*.

Tiedong Liu and Bryan Kian Hsiang Low. 2023. Goat: Fine-tuned llama outperforms gpt-4 on arithmetic tasks.

Shayne Longpre, Le Hou, Tu Vu, Albert Webson, Hyung Won Chung, Yi Tay, Denny Zhou, Quoc V Le, Barret Zoph, Jason Wei, et al. 2023. The flan collection: Designing data and methods for effective instruction tuning. *arXiv preprint arXiv:2301.13688*.

Pan Lu, Baolin Peng, Hao Cheng, Michel Galley, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, and Jianfeng Gao. 2023. Chameleon: Plug-and-play compositional reasoning with large language models. *arXiv preprint arXiv:2304.09842*.

Pan Lu, Liang Qiu, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, Tanmay Rajpurohit, Peter Clark, and Ashwin Kalyan. 2022a. Dynamic prompt learning via policy gradient for semi-structured mathematical reasoning. *arXiv preprint arXiv:2209.14610*.

Pan Lu, Liang Qiu, Wenhao Yu, Sean Welleck, and Kai-Wei Chang. 2022b. A survey of deep learning for mathematical reasoning. *arXiv preprint arXiv:2212.10535*.

Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, et al. 2021. Show your work: Scratchpads for intermediate computation with language models. *arXiv preprint arXiv:2112.00114*.

OpenAI. 2022. Chatgpt.

OpenAI. 2023. Gpt-4 technical report.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744.

Bhargavi Paranjape, Scott Lundberg, Sameer Singh, Hannaneh Hajishirzi, Luke Zettlemoyer, and Marco Tulio Ribeiro. 2023. Art: Automatic multi-step reasoning and tool-use for large language models.

Aaron Parisi, Yao Zhao, and Noah Fiedel. 2022. Talm: Tool augmented language models.

Arkil Patel, Satwik Bhattamishra, and Navin Goyal. 2021. Are nlp models really able to solve simple math word problems? In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2080–2094.

Baolin Peng, Chunyuan Li, Pengcheng He, Michel Galley, and Jianfeng Gao. 2023. Instruction tuning with gpt-4.

Jonas Pfeiffer, Andreas Rücklé, Clifton Poth, Aishwarya Kamath, Ivan Vulić, Sebastian Ruder, Kyunghyun Cho, and Iryna Gurevych. 2020. Adapterhub: A framework for adapting transformers. In *Proceedings of the 2020 Conference on Empirical Methods* *in Natural Language Processing: System Demonstrations*, pages 46–54.

Cheng Qian, Chi Han, Yi R. Fung, Yujia Qin, Zhiyuan Liu, and Heng Ji. 2023. Creator: Disentangling abstract and concrete reasonings of large language models through tool creation.

Yujia Qin, Shengding Hu, Yankai Lin, Weize Chen, Ning Ding, Ganqu Cui, Zheni Zeng, Yufei Huang, Chaojun Xiao, Chi Han, Yi Ren Fung, Yusheng Su, Huadong Wang, Cheng Qian, Runchu Tian, Kunlun Zhu, Shihao Liang, Xingyu Shen, Bokai Xu, Zhen Zhang, Yining Ye, Bowen Li, Ziwei Tang, Jing Yi, Yuzhang Zhu, Zhenning Dai, Lan Yan, Xin Cong, Yaxi Lu, Weilin Zhao, Yuxiang Huang, Junxi Yan, Xu Han, Xian Sun, Dahai Li, Jason Phang, Cheng Yang, Tongshuang Wu, Heng Ji, Zhiyuan Liu, and Maosong Sun. 2023. Tool learning with foundation models.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2023. Hugginggpt: Solving ai tasks with chatgpt and its friends in huggingface. *arXiv preprint arXiv:2303.17580*.

Shelly Sheynin, Oron Ashual, Adam Polyak, Uriel Singer, Oran Gafni, Eliya Nachmani, and Yaniv Taigman. 2022. Knn-diffusion: Image generation via large-scale retrieval. *arXiv preprint arXiv:2204.02849*.

Aarohi Srivastava, Abhinav Rastogi, Abhishek Rao, Abu Awal Md Shoeb, Abubakar Abid, Adam Fisch, Adam R. Brown, Adam Santoro, Aditya Gupta, Adrià Garriga-Alonso, Agnieszka Kluska, Aitor Lewkowycz, Akshat Agarwal, Alethea Power, Alex Ray, Alex Warstadt, Alexander W. Kocurek, Ali Safaya, Ali Tazarv, Alice Xiang, Alicia Parrish, Allen Nie, Aman Hussain, Amanda Askell, Amanda Dsouza, Ambrose Slone, Ameet Rahane, Anantharaman S. Iyer, Anders Andreassen, Andrea Madotto, Andrea Santilli, Andreas Stuhlmüller, Andrew Dai, Andrew La, Andrew Lampinen, Andy Zou, Angela Jiang, Angelica Chen, Anh Vuong, Animesh Gupta, Anna Gottardi, Antonio Norelli, Anu Venkatesh, Arash Gholamidavoodi, Arfa Tabassum, Arul Menezes, Arun Kirubarajan, Asher Mullokandov, Ashish Sabharwal, Austin Herrick, Avia Efrat, Aykut Erdem, Ayla Karakaş, B. Ryan Roberts, Bao Sheng Loe, Barret Zoph, Bartłomiej Bojanowski, Batuhan Özyurt, Behnam Hedayatnia, Behnam Neyshabur, Benjamin Inden, Benno Stein, Berk Ekmekci, Bill Yuchen Lin, Blake Howald, Cameron Diao, Cameron Dour, Catherine Stinson, Cedrick Argueta, César Ferri Ramírez, Chandan Singh, Charles Rathkopf, Chenlin Meng, Chitta Baral, Chiyu Wu, Chris Callison-Burch, Chris Waites, Christian Voigt, Christopher D. Manning, Christopher Potts, Cindy Ramirez, Clara E. Rivera, Clemencia Siro, Colin Raffel, Courtney Ashcraft, Cristina Garbacea, Damien Sileo, Dan Garrette, Dan Hendrycks, Dan Kilman, Dan Roth, Daniel Freeman, Daniel Khashabi, Daniel Levy, Daniel Moseguí González, Danielle Perszyk, Danny Hernandez, Danqi Chen, Daphne Ippolito, Dar Gilboa, David Dohan, David Drakard, David Jurgens, Debajyoti Datta, Deep Ganguli, Denis Emelin, Denis Kleyko, Deniz Yuret, Derek Chen, Derek Tam, Dieuwke Hupkes, Diganta Misra, Dilyar Buzan, Dimitri Coelho Mollo, Diyi Yang, Dong-Ho Lee, Ekaterina Shutova, Ekin Dogus Cubuk, Elad Segal, Eleanor Hagerman, Elizabeth Barnes, Elizabeth Donoway, Ellie Pavlick, Emanuele Rodola, Emma Lam, Eric Chu, Eric Tang, Erkut Erdem, Ernie Chang, Ethan A. Chi, Ethan Dyer, Ethan Jerzak, Ethan Kim, Eunice Engefu Manyasi, Evgenii Zheltonozhskii, Fanyue Xia, Fatemeh Siar, Fernando Martínez-Plumed, Francesca Happé, Francois Chollet, Frieda Rong, Gaurav Mishra, Genta Indra Winata, Gerard de Melo, Germán Kruszewski, Giambattista Parascandolo, Giorgio Mariani, Gloria Wang, Gonzalo Jaimovitch-López, Gregor Betz, Guy Gur-Ari, Hana Galijasevic, Hannah Kim, Hannah Rashkin, Hannaneh Hajishirzi, Harsh Mehta, Hayden Bogar, Henry Shevlin, Hinrich Schütze, Hiromu Yakura, Hongming Zhang, Hugh Mee Wong, Ian Ng, Isaac Noble, Jaap Jumelet, Jack Geissinger, Jackson Kernion, Jacob Hilton, Jaehoon Lee, Jaime Fernández Fisac, James B. Simon, James Koppel, James Zheng, James Zou, Jan Kocoń, Jana Thompson, Jared Kaplan, Jarema Radom, Jascha Sohl-Dickstein, Jason Phang, Jason Wei, Jason Yosinski, Jekaterina Novikova, Jelle Bosscher, Jennifer Marsh, Jeremy Kim, Jeroen Taal, Jesse Engel, Jesujoba Alabi, Jiacheng Xu, Jiaming Song, Jillian Tang, Joan Waweru, John Burden, John Miller, John U. Balis, Jonathan Berant, Jörg Frohberg, Jos Rozen, Jose Hernandez-Orallo, Joseph Boudeman, Joseph Jones, Joshua B. Tenenbaum, Joshua S. Rule, Joyce Chua, Kamil Kanclerz, Karen Livescu, Karl Krauth, Karthik Gopalakrishnan, Katerina Ignatyeva, Katja Markert, Kaustubh D. Dhole, Kevin Gimpel, Kevin Omondi, Kory Mathewson, Kristen Chiafullo, Ksenia Shkaruta, Kumar Shridhar, Kyle McDonell, Kyle Richardson, Laria Reynolds, Leo Gao, Li Zhang, Liam Dugan, Lianhui Qin, Lidia Contreras-Ochando, Louis-Philippe Morency, Luca Moschella, Lucas Lam, Lucy Noble, Ludwig Schmidt, Luheng He, Luis Oliveros Colón, Luke Metz, Lütfi Kerem Şenel, Maarten Bosma, Maarten Sap, Maartje ter Hoeve, Maheen Farooqi, Manaal Faruqui, Mantas Mazeika, Marco Baturan, Marco Marelli, Marco Maru, Maria Jose Ramírez Quintana, Marie Tolkiehn, Mario Giulianelli, Martha Lewis, Martin Potthast, Matthew L. Leavitt, Matthias Hagen, Mátyás Schubert, Medina Orduna Baitemirova, Melody Arnaud, Melvin McElrath, Michael A. Yee, Michael Cohen, Michael Gu, Michael Ivanitskiy, Michael Starritt, Michael Strube, Michał Swędrowski, Michele Bevilacqua, Michihiro Yasunaga, Mihir Kale, Mike Cain, Mimee Xu, Mirac Suzgun, Mo Tiwari, Mohit Bansal, Moin Aminnaseri, Mor Geva, Mozhdeh Gheini, Mukund Varma T, Nanyun Peng, Nathan Chi, Nayeon Lee, Neta Gur-Ari Krakover, Nicholas Cameron, Nicholas Roberts, Nick Doiron, Nikita Nangia, Niklas Deckers, Niklas Muennighoff, Nitish Shirish Keskar, Niveditha S. Iyer, Noah Constant, Noah Fiedel, Nuan Wen, Oliver Zhang, Omar Agha, Omar Elbaghdadi, Omer Levy, Owain Evans, Pablo Antonio Moreno Casares, Parth Doshi, Pascale Fung, Paul Pu Liang, Paul Vicol, Pegah Alipoormolabashi, Peiyuan Liao, Percy Liang, Peter Chang, Peter Eckersley, Phu Mon Htut, Pinyu Hwang, Piotr Miłkowski, Piyush Patil, Pouya Pezeshkpour, Priti Oli, Qiaozhu Mei, Qing Lyu, Qinlang Chen, Rabin Banjade, Rachel Etta Rudolph, Raefer Gabriel, Rahel Habacker, Ramón Risco Delgado, Raphaël Millière, Rhythm Garg, Richard Barnes, Rif A. Saurous, Riku Arakawa, Robbe Raymaekers, Robert Frank, Rohan Sikand, Roman Novak, Roman Sitelew, Ronan LeBras, Rosanne Liu, Rowan Jacobs, Rui Zhang, Ruslan Salakhutdinov, Ryan Chi, Ryan Lee, Ryan Stovall, Ryan Teehan, Rylan Yang, Sahib Singh, Saif M. Mohammad, Sajant Anand, Sam Dillavou, Sam Shleifer, Sam Wiseman, Samuel Gruetter, Samuel R. Bowman, Samuel S. Schoenholz, Sanghyun Han, Sanjeev Kwatra, Sarah A. Rous, Sarik Ghazarian, Sayan Ghosh, Sean Casey, Sebastian Bischoff, Sebastian Gehrmann, Sebastian Schuster, Sepideh Sadeghi, Shadi Hamdan, Sharon Zhou, Shashank Srivastava, Sherry Shi, Shikhar Singh, Shima Asaadi, Shixiang Shane Gu, Shubh Pachchigar, Shubham Toshniwal, Shyam Upadhyay, Shyamolima, Debnath, Siamak Shakeri, Simon Thormeyer, Simone Melzi, Siva Reddy, Sneha Priscilla Makini, Soo-Hwan Lee, Spencer Torene, Sriharsha Hatwar, Stanislas Dehaene, Stefan Divic, Stefano Ermon, Stella Biderman, Stephanie Lin, Stephen Prasad, Steven T. Piantadosi, Stuart M. Shieber, Summer Misherghi, Svetlana Kiritchenko, Swaroop Mishra, Tal Linzen, Tal Schuster, Tao Li, Tao Yu, Tariq Ali, Tatsu Hashimoto, Te-Lin Wu, Théo Desbordes, Theodore Rothschild, Thomas Phan, Tianle Wang, Tiberius Nkinyili, Timo Schick, Timofei Kornev, Timothy Telleen-Lawton, Titus Tunduny, Tobias Gerstenberg, Trenton Chang, Trishala Neeraj, Tushar Khot, Tyler Shultz, Uri Shaham, Vedant Misra, Vera Demberg, Victoria Nyamai, Vikas Raunak, Vinay Ramasesh, Vinay Uday Prabhu, Vishakh Padmakumar, Vivek Srikumar, William Fedus, William Saunders, William Zhang, Wout Vossen, Xiang Ren, Xiaoyu Tong, Xinran Zhao, Xinyi Wu, Xudong Shen, Yadollah Yaghoobzadeh, Yair Lakretz, Yangqiu Song, Yasaman Bahri, Yejin Choi, Yichi Yang, Yiding Hao, Yifu Chen, Yonatan Belinkov, Yu Hou, Yufang Hou, Yuntao Bai, Zachary Seid, Zhuoye Zhao, Zijian Wang, Zijie J. Wang, Zirui Wang, and Ziyi Wu. 2022. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models.

Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford alpaca: An instruction-following llama model. https://github.com/tatsu-lab/stanford_alpaca.

Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, YaGuang Li, Hongrae Lee, Huaixiu Steven Zheng, Amin Ghafouri, Marcelo Menegali, Yanping Huang, Maxim Krikun, Dmitry Lepikhin, James Qin, Dehao Chen, Yuanzhong Xu, Zhifeng Chen, Adam Roberts, Maarten Bosma, Vincent Zhao, Yanqi Zhou, Chung-Ching Chang, Igor Krivokon, Will Rusch, Marc Pickett, Pranesh Srinivasan, Laichee Man, Kathleen Meier-Hellstern, Meredith Ringel Morris, Tulsee Doshi, Renelito Delos Santos, Toju Duke, Johnny Soraker, Ben Zevenbergen, Vinodkumar Prabhakaran, Mark Diaz, Ben Hutchinson, Kristen Olson, Alejandra Molina, Erin Hoffman-John, Josh Lee, Lora Aroyo, Ravi Rajakumar, Alena Butryna, Matthew Lamm, Viktoriya Kuzmina, Joe Fenton, Aaron Cohen, Rachel Bernstein, Ray Kurzweil, Blaise Aguera-Arcas, Claire Cui, Marian Croak, Ed Chi, and Quoc Le. 2022. Lamda: Language models for dialog applications.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. Llama: Open and efficient foundation language models.

Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. 2022. Interleaving retrieval with chain-of-thought reasoning for knowledge-intensive multi-step questions. *arXiv preprint arXiv:2212.10509*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed H Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*.

Chenfei Wu, Shengming Yin, Weizhen Qi, Xiaodong Wang, Zecheng Tang, and Nan Duan. 2023. Visual chatgpt: Talking, drawing and editing with visual foundation models. *arXiv preprint arXiv:2303.04671*.

Renrui Zhang, Jiaming Han, Aojun Zhou, Xiangfei Hu, Shilin Yan, Pan Lu, Hongsheng Li, Peng Gao, and Yu Qiao. 2023. Llama-adapter: Efficient fine-tuning of language models with zero-init attention. *arXiv preprint arXiv:2303.16199*.

Appendices

A Prompt Pattern for ChatGPT Toolkit

We show the pattern of the prompt we apply for the creation of toolkits leveraging GPT-3.5-turbo in Figure 4. The temperature is set to 0.3 to ensure the model clearly follows the instructions while retaining its creativity to a certain extent. The max length during generation is set to 1024. The prompt shown mainly consists of the instruction for toolkit creation, the demonstration of format, sample public data, and the current tasks' meta information.

B Toolkits for tasks from BIG-bench

We show in Figures 6 to 13 the toolkits that GPT-3.5-turbo created leveraging the prompt mentioned in the previous section. Notice that we show the final version of the toolkit, which may contain certain modifications based on human feedback. For instance, in Figure 8, we have integrated addition, subtraction, and hadamard operation into one single tool, as all of them do not change the shape of the given matrix. This will effectively reduce the redundant tools and help the model learn with ease.

C Settings for Chain-of-Solving

C.1 Choice of Instruction

To inspire the models' ability to plan and call the tools during chain-of-solving (CoS), we apply clear instructions to prompt the model. For CoS-planning, we choose the instruction "*You are presented with a question and several tools that may be useful. Select the useful tools and plan how to solve the problem.*", while for CoS-calling, we choose the instruction "*Use the tool given in the input to write code to solve the problem.*". This applies to all the settings, even for the LLaMA-CoS because it is also tuned in an instruction-following way.

C.2 Details about Demonstrations

For all the experiments leveraging ChatGPT, despite the instructions, we also provide the model with demonstration examples to showcase the format of planning and calling, as well as better leveraging its potential. The temperature is set to 0.3 during generation, and the max output length is set to 1024.

For the raw LLaMA-7B and Alpaca baselines without being tuned, the demonstration examples are also applied to provide guidance, while the LLaMA-CoS tuned under our Toolink framework does not need demonstration examples as it is already tuned under the instruction-following paradigm.

D Separating CoS-Planning and CoS-Calling

In Toolink, planning and calling are combined as a whole CoS process, where the plans generated by the model are again fed back to itself to help guide the generation of the final calling decision. To disentangle their functions and better understand their role, we employ tests to measure their accuracy separately.

D.1 CoS-Planning Details

For the CoS-planning test, we provide the model with the instruction and all the available tools in the toolkit. In Figure 5, we showcase the format of the CoS-planning prompt given to the model.

However, plans are generated in the form of natural language, whose accuracy is hard to measure. For simplicity, we instead only measure if the correct tools are called upon to solve the given problem.

Suppose $K_T = \{k_1, k_2, ..., k_N\}$ is the toolkit with N tools for task T. For a specific query, we denote the set of useful tools as $K_{\texttt{use}} \subseteq K_T$ and other redundant tools as $K_{\texttt{rdt}} \subseteq K_T$. Suppose the set of tools called upon during planning is $K_{\texttt{call}} \subseteq K_T$, then the correct tools called is denoted as $K_{\texttt{correct}} = K_{\texttt{call}} \cap K_{\texttt{use}}$, and the erroneous tools called $K_{\texttt{err}} = K_{\texttt{call}} \cap K_{\texttt{rdt}}$. These are the exact definition of the variables that we apply in Equation 5.

If all the useful tools are called correctly and precisely, where $K_{\texttt{call}} = T_{\texttt{use}}$, the accuracy will be 1.00. Note that this metric is relatively strict because wrong calls will result in the deduction of accuracy.

D.2 CoS-Calling Details

For the CoS-calling test, the standard (correct) plans will be provided to the model, instead of the plans that the model previously generated. The CoS-calling test solely aims to investigate the model's ability to follow plans and generate the correct calling decisions. Besides the plans and instructions, only the useful tools with respect to the given query are provided in the prompt, instead

```
### Instruction:
You are asked to create a toolkit to solve the given task. Each tool in the toolkit should be a
Python function that is helpful to solve the problems in the task.
You will be given the related information about the task and the example queries to help you create
the useful tools.
For each tool you create, please first state the tool’s name and its purpose, and then give its
implementation using python code. Please wrap the code in ```python … ```. You can create 2 to 5
tools for each task.
```

```
### Task 1
- Task Meta Information
The task asks the model to unscramble letters into a word. You should arrange the letters in correct
order and output a valid English word.
- Sample Queries
1. Q: Please unscramble the letters into a word, and write that word: imitedl  A: limited
2. Q: Please unscramble the letters into a word, and write that word: ermelonwat  A: watermelon
3. Q: Please unscramble the letters into a word, and write that word: ttlebo   A: bottle
Please create a toolkit including Python functions that are useful in solving the problem.
### Response
- Tool 1:
permutations: this tool takes in a given string, generates all possible permutations of it, and
returns them in a list.
```python
import itertools
def permutations(word):
    perms = [''.join(p) for p in itertools.permutations(word)]
    return perms
```
- Tool 2:
valid_word_lookup: this tool takes in a string, checks whether it’s a valid English word, and returns
a bool value.
```python
import nltk
from nltk.corpus import words
def valid_word_lookup(word):
    nltk.download('words')
    english_words = set(words.words())
    return word.lower() in english_words
```
```

```
### Task 2
- Task Meta Information
[Meta Information of the Target Task]
- Sample Queries
[Sample Queries for Toolkit Generation in QA Format]
Please create a toolkit including Python functions that are useful in solving the problem.
### Response
```

Figure 4: The pattern of the prompt given to GPT-3.5-turbo to generate the toolkit.

of all the tools from the toolkit. We showcase the format of the prompt given to the model in Figure 5.

The accuracy of CoS-calling is based on the matching of the model's output to the correct answer. For tasks Arithmetic and Chinese Remainder, the accuracy is evaluated in numerical format; for Matrix Shape, the accuracy is evaluated based on the matching of dimensions list; for all other tasks from BIG-bench, the accuracy is based on the matching of strings between the model's output and the correct answer.

E Dataset Construction

In this section, we provide more details about how CoS-GPT is constructed. We introduce respectively the construction of tool-using data (including planning and calling) and code generation data. All the data points aim to enhance the open-source model's CoS ability.

E.1 Construction of Tool-Using Data

For each query in AQUA-RAT, GSM8K, and TabMWP, we first utilize ChatGPT to create a diverse set of tools that are potentially relevant to the given query, forming the toolkit. We then provide this toolkit to ChatGPT and allow it to select the most suitable tools. Subsequently, we prompt ChatGPT to generate decision calls based on the selected tools and manually verify the correctness of the resulting outputs. If the final answer is correct,

```
Prompt Format for Tool Plan:
### Instruction:
You are presented with a question and several tools that may be useful. Select the useful tools and
plan how to solve the problem.
### Input:
- Question:
[Query from data]
- Available Tools:
1. [Name: Introduction about purpose, inputs, outputs]
2. [Name: Introduction about purpose, inputs, outputs]
...
### Response:
```

```
Prompt Format for Tool Call:
### Instruction:
[Query from data]
Use the tool given in the input to write python code to solve the problem.
### Input:
- Tool 1:
[Name: Introduction about purpose, inputs, outputs]
[Simplified Code Realization]
- Tool 2:
[Name: Introduction about purpose, inputs, outputs]
[Simplified Code Realization]
...
- Plan
[Plan from Model's Tool Plan Response or the Standard (Correct) Plan]
### Response:
```

Figure 5: The format of the data (and prompt) for CoS-planning and CoS-calling.

we divide ChatGPT's responses into two distinct components, representing the planning step and the calling step, which are then individually added to the dataset. In this manner, the validity of our data points can thus be guaranteed.

Throughout these steps of data construction, we also incorporate demonstration examples sampled from the constructed dataset, thereby expanding the dataset in a self-iterative manner. Figure 5 show detailed information about the format of the query. Besides the query, we also provide the corresponding CoS-planning or CoS-calling response and the implementation of the toolkit with useful and redundant tools.

E.2 Construction of Code Generation Data

The code generation data in CoS-GPT are sourced from 6 different venues, including Python-Simple, Python-Specific, Math, Algorithm, LeetCode, and Rectification. The objective behind these categories is to enhance the model's proficiency in problem-solving through code utilization, calling existing packages, applying reasoning, employing algorithms, completing codes of challenging competitions, and engaging in self-rectification.

For Python-Simple and Python-Specific, the former aims to boost the models' ability to solve simple problems using codes, while the latter aims to enhance the model's ability to leverage code packages to solve more complex problems. Both these two sets are generated using ChatGPT. We prompt the model with instructions and demonstrations and gather the code snippets the model generated to solve the given problem.

The queries for the Math set are sampled from the training set of MathQA (Amini et al., 2019) and augmented with a code solution based on the given query and reasoning, leveraging ChatGPT. The generated programs are verified to ensure the output answer is the same as the correct one originally, thus ensuring the validity of the augmented data points. The Algorithm set is extracted from the open-source Python algorithm repository, with over 40 categories and more than a hundred diverse algorithms. For each algorithm, we ask ChatGPT to generate a query related to it and use a code snippet to solve the problem. The codes and corresponding queries are then gathered and formed into the instruction-following format.

For the LeetCode set, we directly extract the official open-sourced problems and the code answers from the website and form our data. The Rectification set is gathered from the error codes generated in the five sets before. The error tracebacks and the bad code snippet is fed into ChatGPT, and we leverage it to rectify the codes and generate a correct code snippet that can solve the given query successfully. We gather the generated codes and execute them again, retaining only the ones that give a correct answer finally and form the set based on these valid data points.

F Main Experiment Setting Details

For our main experiment, we finetune the LLaMA-7B model on four A100-80G GPUs, with a total batch size of 32 and a learning rate of 1e-5. For the model whose performance we demonstrate in Tables 3 and 4, its training dataset consists of 1.6K target task-specific data points (8 tasks, 100 for planning and 100 for calling each), 4K tool-using data and 3K code-generation data randomly sampled respectively from the CoS-GPT dataset. We trained the LLaMA-7B on these data for 3 epochs and obtain LLaMA-CoS.

In addition, for the ablation study about the training on codes we perform in Section 4.3, we apply 7K tool-using data and remove all the code-generation data points. We keep all the other settings the same in this study.

G Diverse CoS Patterns Case Study

In Figure 16, we present three case studies highlighting the diverse nature of LLaMA-CoS in applying planning and calling for tool-using.

Firstly, LLaMA-CoS exhibits the ability to generate sequential plans involving different tools. In the first case, the model simulates the operation on matrices step by step in a linear way and finally gets the correct result.

Secondly, LLaMA-CoS demonstrates proficiency in executing complex tool calls within branch-loop structures. In the second case, the model learns to use different stack operations based on the character met in the expression, and can call the useful tool in a loop structure.

Lastly, the model showcases its competence in performing nested tool invocations. In the third case, the model is able to directly pass the converted hour retrieved from the previous tool as the input parameter for the next tool, which illustrates a successful nested tool call.

These examples serve to show the robustness, versatility, and adaptability of LLaMA-CoS across a wide range of scenarios.

H CoS on Generic Toolkit Details

We source two new tasks, Dynamic Counting and Unit Interpretation, from the BIG-bench. We apply all the problems in Dynamic Counting for our test of toolkit generalization. However, for Unit Interpretation, we specifically select the data from LV 1 in order for the tools from task Arithmetic can be properly applied. To ensure fairness, we expand the dataset by interactively sampling new questions with similar patterns from ChatGPT and incorporating them until the dataset reaches its original full size. Note that we only aim to showcase the toolkit's generalization ability and compare the performance of LLaMA-CoS and ChatGPT within this paper, so we deem expanding the dataset as fair and reasonable under our settings.

We show the toolkits specially tailored for these two new tasks in Figures 14 and 15. The LLaMA-CoS model we apply is still the model we have trained in the main experiment, detailed in Appendices F. All the other settings, including the ChatGPT applied under our framework, are kept the same as that in the main experiment.

Toolkit for task: Arithmetic
- **Tool 1:**
add: it takes in two numbers and returns their sum
```python
def add(a, b):
    return a + b
```
- **Tool 2:**
sub: it takes in two numbers a and b and returns a - b
```python
def sub(a, b):
    return a - b
```
- **Tool 3:**
mul: it takes in two numbers and returns their product
```python
def mul(a, b):
    return a * b
```
- **Tool 4:**
div: it takes in two numbers a and b and returns the integer value of a / b
```python
def div(a, b):
    return int(a / b)
```
- **Tool 5:**
mod: it takes in two numbers a and b and returns a % b
```python
def mod(a, b):
    return a % b
```

Figure 6: The toolkit for task Arithmetic.

```
Toolkit for task: Date Understanding
-  Tool 1:
add_time: It takes in the start day in format MM/DD/YYYY, and calculate the date after y years, m
months and d days. It returns a string in format MM/DD/YYYY.
```python
import datetime
def add_time(start_day, years=0, months=0, days=0):
    start_date = datetime.datetime.strptime(start_day, "%m/%d/%Y")
    new_date = start_date + datetime.timedelta(days=days)
    if new_date.month + months > 12:
        r = int((new_date.month + months) / 12)
        new_date = new_date.replace(year=new_date.year + years + r, month=(new_date.month + months -
           1) % 12 + 1)
    else:
        new_date = new_date.replace(year=new_date.year + years, month=new_date.month + months)
    return new_date.strftime("%m/%d/%Y")
```
- Tool 2:
subtract_time: It takes in the start day in format MM/DD/YYYY, and calculate the date y years, m
months and d days before this day. It returns a string in format MM/DD/YYYY.
```python
import datetime
def subtract_time(start_day, years=0, months=0, days=0):
    start_date = datetime.datetime.strptime(start_day, "%m/%d/%Y")
    new_date = start_date - datetime.timedelta(days=days)
    if new_date.month - months <= 0:
        r = int((new_date.month - months) / -12) + 1
        new_date = new_date.replace(year=new_date.year - years - r, month=(new_date.month - months -
           1) % 12 + 1)
    else:
        new_date = new_date.replace(year=new_date.year - years, month=new_date.month - months)
    return new_date.strftime("%m/%d/%Y")
```
- Tool 3:
convert_hour: It takes the number of hours and convert it into days (integer).
```python
import math
def convert_hour(hours):
    days = math.ceil(hours / 24)
    return days
```
```

Figure 7: The toolkit for task Date Understanding.

Toolkit for task: Matrix Shape

- **Tool 1:**

multiply: it takes in two lists representing the shape of two matrix, and returns the shape of their product.

```python
def multiply(shape1, shape2):
    if shape1[1] != shape2[0]:
        raise ValueError("Matrix shapes are not compatible for multiplication.")
    result_shape = shape1[:-1] + [shape2[-1]]
    return result_shape
```

- **Tool 2:**

kronecker: it takes in two list representing the shape of two matrix, and returns the shape of their kronecker product.

```python
def kronecker(shape1, shape2):
    if len(shape1) != len(shape2):
        raise Exception("The number of dimensions of the two matrices is not equal")
    result_shape = [dim1 * dim2 for dim1, dim2 in zip(shape1, shape2)]
    return result_shape
```

- **Tool 3:**

sum_over_axis: it takes a list representing the shape of the matrix, and the dimension of the axis that is to be sum up. It returns the shape of the resulting matrix.

```python
def sum_over_axis(shape, axis):
    if axis >= len(shape):
        raise ValueError("Invalid axis dimension.")

    result_shape = shape[:axis] + shape[axis+1:]
    return result_shape
```

- **Tool 4:**

transpose: it takes a list representing the shape of a matrix to be transposed, and returns the shape of the resulting matrix.

```python
def transpose(shape):
    result_shape = list(reversed(shape))
    return result_shape
```

- **Tool 5:**

add_subtract_hadamard: it takes two lists representing the shape of two matrices for add, sbstract and hadamard, and returns the shape of the resulting matrix.

```python
def add_subtract_hadamard(shape1, shape2):
    assert shape1 == shape2
    return shape1
```

Figure 8: The toolkit for task Matrix Shape.

````
Toolkit for task: Navigation
-  Tool 1:
update_orientation: It takes the original orientation(N, E, S or W) and turn direction(left, right or
around), and returns the new orientation. It should be used only if not always face forward.
```python
def update_orientation(orientation, turn_direction):
    orientations = ["N", "E", "S", "W"]
    current_index = orientations.index(orientation)
    if turn_direction == "left":
        new_index = (current_index - 1) % 4
    elif turn_direction == "right":
        new_index = (current_index + 1) % 4
    elif turn_direction == "around":
        new_index = (current_index + 2) % 4
    else:
        raise ValueError("Invalid turn direction.")
    return orientations[new_index]
```
- Tool 2:
update_location: It takes the current location(x, y), orientation(N, E, S or W), and steps, and
returns the new location after action.
```python
def update_location(current_location, orientation, steps):
    x, y = current_location
    if orientation == "N":
        new_location = (x, y + steps)
    elif orientation == "E":
        new_location = (x + steps, y)
    elif orientation == "S":
        new_location = (x, y - steps)
    elif orientation == "W":
        new_location = (x - steps, y)
    return new_location
```
````

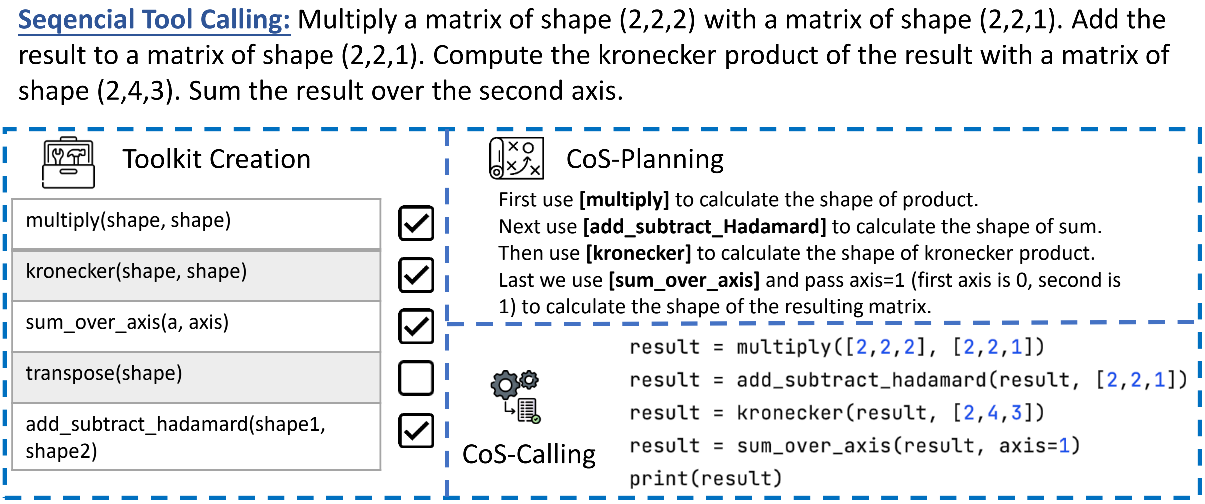
Figure 9: The toolkit for task Navigation.

````
Toolkit for task: Chinese Remainder
-  Tool 1:
divide_remain: it takes in a, b, and c, and checks if the remainder of a divided by b is equal to c.
```python
def divide_remain(a, b, c):
    return a % b == c
```
- Tool 2:
check_validity: it takes into a list of possible answers, and filters the list of answers based on
the upper bound x.
```python
def check_validity(answers, x):
    return [answer for answer in answers if answer <= x]
```
````

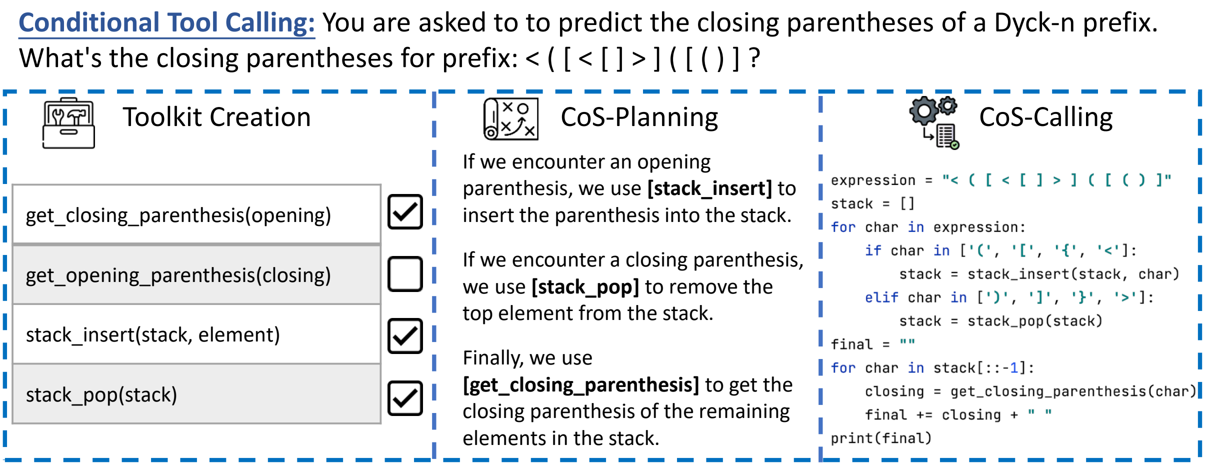
Figure 10: The toolkit for task Chinese Remainder.

```
Toolkit for task: Dyck Language
- Tool 1:
get_closing_parenthesis: This tool takes in an opening parenthesis and returns the corresponding
closing parenthesis.
```python
def get_closing_parenthesis(opening):
    openings = ['(', '[', '{', '<']
    closings = [')', ']', '}', '>']
    if opening in openings:
        return closings[openings.index(opening)]
    else:
        return None
```
- Tool 2:
get_opening_parenthesis: This tool takes in an closing parenthesis and returns the corresponding
opening parenthesis.
```python
def get_opening_parenthesis(closing):
    openings = ['(', '[', '{', '<']
    closings = [')', ']', '}', '>']
    if closing in closings:
        return openings[closings.index(closing)]
    else:
        return None
```
- Tool 3:
stack_insert: This tool takes in a stack and an element and returns the stack with the element
inserted at the top.
```python
def stack_insert(stack, element):
    stack.append(element)
    return stack
```
- Tool 4:
stack_pop: This tool takes in a stack and returns the stack with the top element removed.
```python
def stack_pop(stack):
    if len(stack) > 0:
        stack.pop()
    return stack
```
```

Figure 11: The toolkit for task Dyck Language.

```
Toolkit for task: Boolean Expression
-  Tool 1:
evaluate_expression: this tool takes in an expression as a string, evaluates it using Python's eval()
function, and returns the result.
```python
def evaluate_expression(expression):
    try:
        result = eval(expression)
        return result
    except SyntaxError:
        return "Invalid expression"
```
- Tool 2:
extract_valid_expressions: this tool takes in a string and extract the valid string that represents
the expression.
```python
def extract_valid_expressions(question_string):
    expression = question_string.split(':')[1].split('is')[0].strip()
    return expression
```
```

Figure 12: The toolkit for task Boolean Expression.

```
Toolkit for task: Tracking Shuffled Objects
-  Tool 1:
create_object_dict: this tool takes in a list of people and their initial object, and returns a
dictionary mapping each person to their object.
```python
def create_object_dict(people, objects):
    object_dict = dict(zip(people, objects))
    return object_dict
```
- Tool 2:
update_object_dict: this tool takes in an object dictionary, a list of object trades, and updates the
object dictionary based on the trades.
```python
def update_object_dict(object_dict, trades):
    for trade in trades:
        person1, person2 = trade.split(' and ')
        object_dict[person1], object_dict[person2] = object_dict[person2], object_dict[person1]
    return object_dict
```
- Tool 3:
parse_trades: this tool takes in a string of trades and returns a list of individual trades.
```python
def update_object_dict(object_dict, trades):
    def parse_trades(trades_str):
    trades = trades_str.split('. Then, ')
    trades[0] = trades[0].replace('At the start', '')
    trades[-1] = trades[-1].replace('At the end', '')
    return trades
```
- Tool 4:
get_final_object: this tool takes in a object dictionary and returns the object held by the target
person finally.
```python
def get_final_object(object_dict, target_person):
    return object_dict[target_person]
```
```

Figure 13: The toolkit for task Tracking Shuffled Objects.

```
Toolkit for task: Dynamic Counting
-  Tool 1:
get_closing_parenthesis: This tool takes in an opening parenthesis and returns the corresponding
closing parenthesis.
```python
def get_closing_parenthesis(opening):
    pairs_open = {'(': ')', '[': ']', '{': '}', '<': '>'}
    if opening in pairs_open:
        return pairs_open[opening]
    else:
        return None
```
- Tool 2:
find_open_remaining: This tool takes in a sequence of parenthesis and returns the single reamining
opening paraenthesis that is not closed.
```python
def find_open_remaining(parenthesis):
    parenthesis = parenthesis.split(" ")
    pairs_open = {'(': ')', '[': ']', '{': '}', '<': '>'}
    pairs_close = {')': '(', ']': '[', '}': '{', '>': '<'}
    stack = []
    for p in parenthesis:
        if p in pairs_open:
            stack.append(p)
        elif p in pairs_close:
            if pairs_close[p] in stack:
                stack.remove(pairs_close[p])
    return stack[0]
```
```

Figure 14: The toolkit for task Dynamic Counting.

````
Toolkit for task: Unit Interpretation (LV1)
-  Tool 1:
same_ratio_calculate_first: This tool assumes that a : b = x : y. x is the first position. It takes
in a, b and y and returns x.
```python
def same_ratio_calculate_first(a, b, y):
    return (a / b) * y
```
- Tool 2:
same_ratio_calculate_second: This tool assumes that a : b = x : y. y is the second position. It takes
in a, b and x and returns y.
```python
def same_ratio_calculate_second(a, b, x):
    return (b / a) * x
```
````

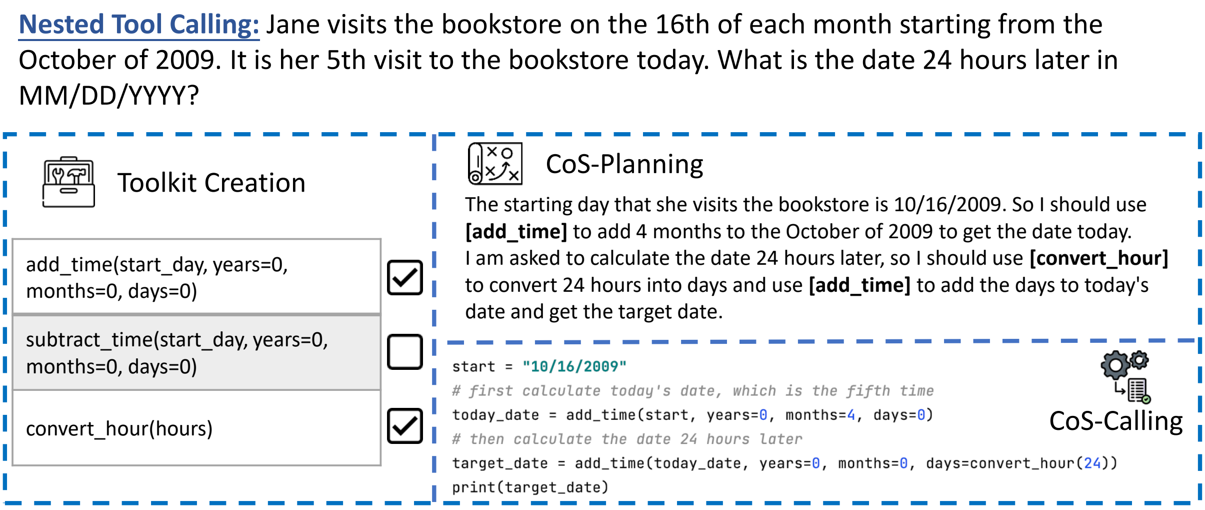
Figure 15: The toolkit for task Unit Interpretation (LV 1).



(a) Sequential Tool Calling.



(b) Conditional Tool Calling.



(c) Nested Tool Calling.

Figure 16: Case Studies on the diverse CoS-calling patterns in the main experiment.