



RTaC: A Generalized Framework for Tooling

Nisarg Bhavsar^(✉), Abhinav Thakur, Amrit Lal Singh,
and Ashish Patwa

Indian Institute of Technology Kharagpur, Kharagpur 721302, India
nisargbhavsar25@kgpian.iitkgp.ac.in
<https://www.iitkgp.ac.in>

Abstract. In the rapidly evolving domain of Large Language Models (LLMs), integrating tool usage remains a formidable challenge, particularly when it comes to the dynamic selection and sequencing of tools in response to complex queries. Addressing this, we introduce Reimagining Tooling as Coding (RTaC), a groundbreaking framework that transforms tool usage into a coding paradigm. Inspired by recent advancements [18], RTaC conceptualizes tools as Python functions within a dual-agent system [2], significantly enhancing LLMs' tool usage efficiency. Our comprehensive experiments reveal that RTaC enables coding-based LLMs, such as DeepSeek and CodeLlama, to achieve and surpass GPT-4 benchmarks in cost-effectiveness and latency without compromising on handling intricate tool sequencing with conditional and iterative logic. This research not only sets a new benchmark for tooling efficiency in LLMs but also opens new avenues for the application of LLMs in complex problem-solving scenarios, heralding a significant leap forward in the functionality and versatility of LLMs across diverse domains.

Keywords: Large Language Models (LLMs) · Dual Agent System · Python Functions for Tool Integration · Automated Tool Sequencing · Advanced LLM Applications · RTaC Framework

1 Introduction

In the evolving landscape of LLMs, their use as reasoning and tooling agents has garnered substantial attention. LLMs demonstrate the capacity to interpret and respond to queries by calling tools [11, 12], a testament to their advanced language comprehension. This capability to integrate tool usage represents a significant stride in enhancing the scope and accuracy of LLMs in various applications. Current state-of-the-art approaches to the tool-usage problem, which utilize GPT-4 (OpenAI) and Claude-2 (Anthropic), demonstrate impressive results but are closed-source and computationally expensive. Researchers have attempted to solve this problem by fine-tuning smaller language models [11, 12, 15]. However, these models are ineffective at generalizing to new tools when provided in a

zero-shot manner, referred to as ‘dynamic tooling’ from here onwards. The discrepancy between the generalized tool-use capabilities of large models and the more restricted capabilities of compact models presents the motivation behind our work - Can we exploit the nature of this task to train small open-source LLMs to generalize their tool-use abilities while keeping the latency minimal? (Fig. 1).

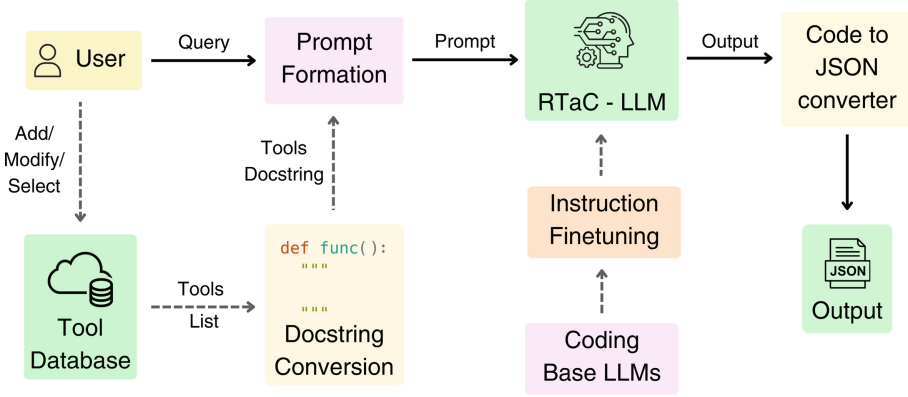


Fig. 1. Overview of “Reimagining Tooling as Coding” (RTaC)

Addressing these challenges, we propose Reimagining Tooling as Coding (RTaC), which reconceptualizes tooling as a code-generation task to exploit the powerful code-comprehension capabilities of LLMs. RTaC provides tools to be used, in docstring format, to instruct fine-tuned coding-base LLMs. It then extracts the output in Python-inspired code format and deterministically converts it to JSON. RTaC promotes docstring reading capability in the LLMs, supporting tool modification, addition, and deletion. We use RTaC to achieve GPT-4 benchmark performance while employing smaller models, such as DeepSeek 1.3B and CodeLlama 7B LLMs, despite a drastic (300x) reduction in parameter count, as shown in Sect. 5. We simultaneously achieve significant (5x) cost reduction per query while matching GPT-4’s latency. Moreover, RTaC supports processing complex conditional and iterative logic, surpassing GPT-4’s capabilities.

2 Related Works

2.1 Dataset and Tooling Benchmarks

Various domain-specific tooling datasets have been proposed like API-Bank [7], ToolEyes [20], RoT-Bench [21], EasyTool [22] and MetaTool [5]. These are domain-specific and assess LLMs’ tool usage and tool-identifying abilities.

- **API-Bank:** Developed from interviews with over 500 users, this benchmark includes a training set created through a multi-agent approach and a diverse

set of manually annotated dialogues to assess LLMs’ API usage across various domains and complexities.

- **ToolEyes:** Features a comprehensive evaluation system with 600+ tools across 7 scenarios, assessing LLMs on five critical dimensions to expose capability gaps and generate research insights.
- **RoT-Bench:** Evaluates LLMs’ ability to accurately select tools, identify parameters, and fill content in environments ranging from noise-free to highly variable real-world conditions.
- **EasyTool:** Addresses issues from inconsistent documentation by creating standardized tool instructions to improve LLMs’ tool usage proficiency.
- **MetaTool:** Assesses LLMs’ tool selection awareness and suitability across various tasks and scenarios, highlighting biases and current limitations.

These benchmarks are domain-specific and unrelated to tool usage as a function-calling approach. We thereby went on to build our dataset for the task using a dual agent system and test our approach on it.

2.2 Tooling LLMs

The application of LLMs for tooling is a profound task, and various research, as mentioned in TALM [10], which uses tools in context to solve different tasks; LATM [2] showed that LLMs can be used to create and reuse different tools created by them in order to act as intelligent Agents. Various tooling LLMs like Tool LLaMA, ToolAlpaca [15], and Gorilla [11] are available and suitable for use as domain-specific agents. However, they are captivated by the out-of-domain tool usage capabilities, identifying the correct set of tools and assigning appropriate arguments to them. Tool Llama is a fine-tuned version of Llama-70B on the ToolBench Dataset, as mentioned in the paper [12]. The model works well on general domain tools but fails in context-dependent scenarios that use tools. ToolAlpaca is a generalized tool LLM that adapts off-domain tools for usage. It is still hard for the LLM to reason on complex tool ordering scenarios.

Recent approaches have aimed to augment LLMs with the ability to utilize tools and resources. TALM [10] introduces a framework for integrating tools with LLMs like T5 via a text-to-text API, enabling generalization to out-of-distribution inputs solvable with access to tools. It employs a policy-gradient reinforcement learning algorithm to fine-tune the LLM for tool usage. The Hugging-GPT system [14] leverages the Hugging Face API to solve AI tasks using LLMs. Toolformer [13] is an LLM pre-trained on an annotated dataset, exhibiting prowess in solving complex problems by leveraging external APIs. However, it is constrained by a fixed set of available tools and the inability to chain tool usage. Toolformer also implements novel self-supervised augmentation during training. These approaches demonstrate the potential of enhancing LLMs with tool utilization capabilities while highlighting challenges such as generalization, tool chaining, and scalability to new tools.

Gorilla [11] stands out as a pivotal work that uses the LLAMA-7B model to accurately extract APIs from repositories like TensorHub, HuggingFace, and

TorchHub. It significantly outperforms GPT-4 in API functionality accuracy and reduces hallucination errors, emphasizing enhancing LLMs’ practical utility over conversational skills. Evaluated on an extensive dataset of 11,000 API pairs, Gorilla demonstrates commendable retrieval capabilities. However, Its reliance on machine learning datasets and the necessity for fine-tuning raise questions about generalizability and adaptability to custom APIs. Complementing Gorilla, ToolBench [12] is a large-scale benchmark containing over 16,000 high-quality APIs across 3,451 tools, facilitating robust evaluation of LLMs’ API usage skills. It employs a 3-stage-construction process, comprehensive API metadata, and diverse instruction generation. GPT-3.5 searches for valid action sequences using accurate API responses through multi-round conversations, leveraging a Depth-First Search Decision Tree (DFS DT) to expand the search space. With its scale, diversity, realism, and expanded search methodology, ToolBench enables a thorough assessment of LLMs’ capabilities in utilizing APIs to accomplish tasks.

Ultimately, we look at Tool Alpaca, a framework to improve compact language models’ generalized tool usage skills. It first constructs a diverse corpus spanning 50 categories and 426 tools with 3938 usage instances generated via multi-agent simulation. This corpus is then used to fine-tune compact Vicuna models, creating ToolAlpaca-7B and 13B. Experiments on unseen simulated and real-world tools demonstrate that ToolAlpaca models achieve strong generalization comparable to large models like GPT-3.5. Tool diversity is shown to be critical, with performance improving as the variety of tools in the corpus increases. Overall, ToolAlpaca provides an automated approach using simulation and diversity to instill generalized tool usage abilities in compact models, enabling them to adapt to new tools.

All these approaches work well, but they depend on the core tools on which they are trained and fine-tuned. Out-of-domain tool usage is a difficult task for all of these models.

2.3 Prompting Methods

Prompting is also a significant method to improve the LLM context adherence capability and can also be used in the region of agentic LLMs. Various prompting methods, such as Chain-of-Thought [17], Tree-of-Thoughts [19], Graph-of-Thought [1], Skeleton-of-Thought [9], and Knowledge Graph [3] addition, can increase a LLM’s overall context understanding capability. One more technique to look at is multi-tool COT Prompting,

Recent research has proposed several innovative frameworks to improve the multi-step reasoning capabilities of LLMs by combining chain-of-thought (CoT) prompting techniques with external tool integration. The “Tree of Thoughts” (ToT) [19] approach frames problem-solving as a search through a tree structure, where each node represents a coherent “thought” or intermediate reasoning step. ToT allows LLMs to explore multiple reasoning paths, generate and evaluate candidate thoughts, and direct the exploration using classical search algorithms like breadth-first and depth-first search. The “Graph of Thoughts” (GoT) [1] framework represents information as an interconnected graph, with

vertices denoting individual pieces of information and connections signifying dependencies between them. This flexible structure enables GoT to integrate outputs from various reasoning paths, analyze complex thought networks, and incorporate feedback loops for iterative improvement of LLM outputs.

Furthermore, the MultiTool-CoT [6] framework facilitates LLMs like GPT-3.5 to leverage multiple tools, such as calculators and knowledge retrievers, by inserting triggers for tool invocation at appropriate steps within the CoT reasoning process. Experiments on numerical and knowledge reasoning datasets demonstrate that MultiTool-CoT significantly outperforms baselines, achieving state-of-the-art accuracy by addressing different error types with different tools, with gains from combining tools exceeding individual tool gains.

These approaches have significantly improved over standard prompting techniques across various tasks, including mathematical reasoning, creative writing, and knowledge-based problems. However, challenges persist, such as token limitations for CoT prompting and potential errors in LLM-generated reasoning processes, highlighting the need for further research in this area to unlock the potential of LLMs in complex problem-solving scenarios fully.

3 Method

3.1 RTaC (Reimagining Tooling as Coding)

RTaC is a novel framework that proposes the conversion of tools into Python functions with proper arguments and tool descriptions in the form of a Pythonic tool docstring, which can be appended to the context of the prompt in order to select and sequence the correct tools, with proper arguments. The paper [18] inspires the technique, which discusses how we can empower the capabilities of an LLM using code. The framework involves the creation of the dataset using a dual agent system that generates query output pairs. These pairs are used for instruction fine-tuning various Coding base LLMs, which act as tooling agents. The application of coding-based LLMs helps adhere to various complex conditional and iterative logics in tooling. Our experiments prove that open-source coding base LLMs are better regarding latency and cost per query than benchmark GPT-4. Coding base LLMs concerning normal LLMs perform better due to their fewer hallucinations and higher context adherence ability.

Dataset Generation. The papers above incorporate data generation as their primary approach for adapting base LLMs for tool usage. Gorilla introduces a comprehensive dataset called APIBench by utilizing Self-Instruct [16], which proposes an automated pipeline to generate large-scale instruction datasets from a small set of seed tasks. First, human experts provide sample instructions and API documentation as context. A language model generates new instructions that plausibly use the APIs, creating instruction-API pairs. A vital benefit of this approach is that it does not require manual effort to label training data. Gorilla uses GPT-4 for this data generation.

The RTaC framework employs distinct datasets to rigorously evaluate its performance. Specifically, the datasets are structured as follows: the Static ToolSet comprises 800 query-output pairs, the Dynamic ToolSet includes 700 pairs, and the Conditional/Iterative ToolSet consists of 300 pairs. Additionally, 200 unanswerable philosophical queries are incorporated to test the model’s robustness in handling queries beyond its configured capabilities (Fig. 2).



Fig. 2. Dual agent dataset generation

Although APIBench is built over massive APIs, it does not have multi-tool scenarios. ToolLLM proposes an innovative data generation strategy supporting multi-tool interplay-the paper samples API combinations by iterating through tools and sampling intra-category and intra-collection combinations. GPT-3.5 is leveraged to generate instructions involving the sampled APIs, and its behavior is regulated by prompting with documentation, task descriptions, and examples. Generated APIs are validated against the original sample to filter out hallucinations.

Static ToolSet. The Generation of the static toolset was done using a set of initial pre-defined tools as shown in Appendix A.3. As shown in the figure, the query output pairs are generated using an agent fed with different tools. As shown below, query templates are generated, which are then randomly filled by GPT-4, and the solution APIs for those are also generated using the GPT-4 Model. Then, the dataset underwent a thorough human evaluation process, which we used for instruction fine-tuning.

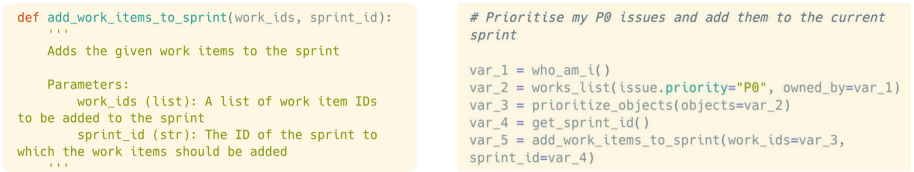
Dynamic ToolSet. The Dynamic toolset, as the name Dynamic suggests, is not pre-defined, and they are generated in the correct format using a different agent. The Dynamic toolset is generated using a procedure when the tools are generated using one agent. Then, we again used GPT-4 as the agent to generate query templates, fill those query templates, and get the relevant set of tool APIs for those queries. These queries then undergo a rigorous human evaluation for syntactic and logical correctness.

Conditional and Iterative ToolSet. Utilizing various tools with conditional and iterative logic requires the creation of particular query output pairs where we use conditional logic like if-else and iterative logic like loops. These kinds of queries are complex to handle by normal LLMs, and they require fine-tuning and an innate logic formation ability present in a coding-based LLM. The query output pair was again generated through GPT-4, and the query templates were filled out using the agent by random values, and then the response was generated. Again, these queries went through a robust human evaluation.

A dataset containing 1800 Query Output pairs was generated with static and dynamic toolsets, and 200 unanswerable philosophical queries were generated; a total of 2000 Query Output pairs were named Stage 1 Dataset. Additionally, 100 query output pairs were generated with conditional and iterative logic, and this was combined with Stage 1 datasets, which were randomly selected to be 500 pool. This dataset was named Stage 2 Dataset. This Whole Dataset was used to do instruction fine-tuning on various LLMs.

Design Framework. Reimagining tooling as a form of coding in the context of LLMs forms the cornerstone of our pipeline design. This approach stems from the observation that tool utilization in LLMs essentially involves executing function calls, assigning values to arguments, and efficiently linking these outputs, mirroring the core elements of coding. This conceptual overlap extends beyond mere theory, as evidenced by the proficiency of Copilot and CodeGen. Grounded in this insight, we adopt a training strategy that treats the tooling challenge within the framework of a coding paradigm. Accordingly, we prioritize fine-tuning LLMs with a foundational background in coding (such as DeepSeek-1.3B Code-Instruct) instead of those exclusively trained on natural language processing tasks.

In our paradigm, tool descriptions are conveyed to LLMs in a docstring format during training, as shown in Fig. 3 (left), emulating standard coding practices. The expected output format is structured as variable assignments from API calls (e.g., `var_x = api_call(arguments)`), as shown in Fig. 3 (right). This format offers advantages over direct training on JSON outputs by reducing the number of output tokens required and circumventing the need for additional training to correct JSON errors, an issue prevalent in other methods [11, 12].



```
def add_work_items_to_sprint(work_ids, sprint_id):
    """
    Adds the given work items to the sprint

    Parameters:
        work_ids (list): A list of work item IDs
        to be added to the sprint
        sprint_id (str): The ID of the sprint to
        which the work items should be added
    """
```

```
# Prioritise my P0 issues and add them to the current
sprint

var_1 = who_am_i()
var_2 = works_list(issue.priority="P0", owned_by=var_1)
var_3 = prioritize_objects(objects=var_2)
var_4 = get_sprint_id()
var_5 = add_work_items_to_sprint(work_ids=var_3,
sprint_id=var_4)
```

Fig. 3. Sample of tool docstring (left) and output in code format (right)

Conditional and iterative logic is handled by allowing the LLM to generate outputs in the format of `var_x = api_calls()` and incorporate if-else statements and for-loop constructs. In the parsed JSON object, we introduce a specialized magic tools - conditional_magic with the capability for 'JSON in JSON' style argument values, as shown in Fig. 4. Such a format is crucial for managing multiple chain tools dependent on specific conditions or requiring iterative processes.

```
# if current sprint is "SPR-123", ring
bell. after that, archive all p0 issues
created by user "USR-123".

var_1 = get_sprint_id()

if var_1 == "SPR-123":
    temp_1 = ring_bell()

var_2 = works_list(issue.priority=["p0"],
created_by=["USR-123"], type=["issue"])

for loop_var in var_2:
    temp_2 = archive_item(work_id=loop_var)
```

```
{
  "tool_name": "get_sprint_id",
  "arguments": []
},
{
  "tool_name": "conditional_magic",
  "condition": "$$PREV[0] == \"SPR-123\"",
  "true": [
    {
      "tool_name": "ring_bell",
      "arguments": []
    }
  ],
  "false": []
},
...
{
  "tool_name": "iterational_magic",
  "looping_var": "$$PREV[2]",
  "loop": [
    {
      "tool_name": "archive_item",
      "arguments": [
        {
          "argument_name": "work_id",
          "argument_value": "$$LOOP_VAR"
        }
      ]
    }
  ]
},
...
```

Fig. 4. Sample code output and conversion using 'JSON in JSON' methodology

Fine Tuning Methods. We propose fine-tuning our LLM using the data generated by the earlier methodology to achieve both input format comprehension and output format adherence. This fine-tuning utilizes Stage 1 and 2 datasets to instill in the LLM docstring comprehension and adherence to our Python-inspired output format.

Training Pipeline: We follow an instruction fine-tuning-based training approach wherein we first prepare our dataset in a structure where an “Allowed tools”: token is introduced, followed by the docstrings for the tools to be used as shown in Appendix A.2. We train the model on the LORA [4] framework using the PEFT library under a 4-bit quantization setting through the Bits and Bytes framework. Training is done in two stages. During the first stage, the model is trained for five epochs on queries from the Stage 1 Dataset and the docstrings for the tools in the Problem Statement. The model is further trained for five epochs using the Stage 2 Dataset, where it sees the docstrings for the tools in the Problem Statement and the five new tools described above. This short instruction fine-tuning instills docstring reading capabilities in the LLM and adherence to our Python-inspired code output format.

Inference: RTaC allows the user to add their own set of dynamic tools. These tools are appended to the static tools in the prompt under the “Allowed Tools”: token and then passed to the LLM. Similarly, updating docstrings are passed under the “Allowed Tools”: token in modifying and deleting already added tools.

3.2 Evaluation Metrics

BLEU score focuses solely on n-gram precision and may not accurately reflect semantic similarity; hence, it is not an optimal metric for evaluating the

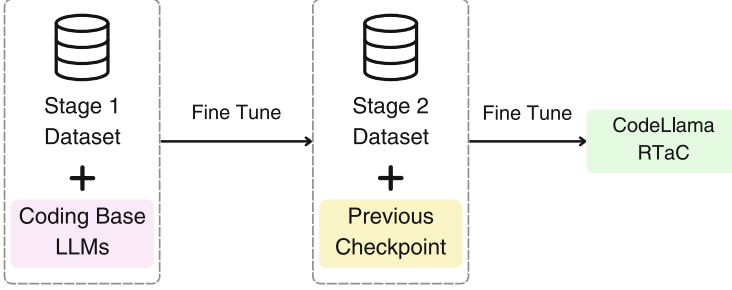


Fig. 5. RTaC Training Pipeline

performance of our approach. Other metrics, such as the JSON Similarity Score and the F1-Score, were used to evaluate the final approach (Fig. 5).

JSON Similarity Score: The JSON Similarity Score is a metric used to quantify the degree of similarity between two JSON objects. It measures how closely the structure and content of two JSON objects align, aiding in data comparison, deduplication, and schema matching. Analyzing key-value pairs, arrays, and nested structures provides insights into the level of resemblance between datasets. This score is precious in data integration processes, ensuring consistency and accuracy across disparate sources. Its ability to assess JSON efficiently facilitates seamless data exchange and interoperability in diverse applications. We use this to assess the correctness of our generated tool structure.

F1-Score: In our evaluation framework, the F1-Score is calculated based on the precision (the proportion of relevant instances among the retrieved instances) and recall (the proportion of relevant instances that were retrieved). The true positives are defined as the set of tools correctly identified and utilized by the LLM, while false positives are those incorrectly predicted tools, and false negatives are correct tools that the model failed to identify. This metric is particularly suited for assessing the accuracy of tool selection and usage in complex LLM operations.

4 Experiments

We have conducted various experiments on open-source coding base, normal, and Closed Source LLMs. The experiments ranged from using retrievers to various prompting methods and fine-tuning approaches on different stage datasets. The Experiments were done on Google colab, where fine-tuning was done on A-100 GPUs, and inference was done on T4 and A100 GPUs. Within the Regime of open source LLMs, we experimented with Llama-7b, Zephyr-7b, Codellama-7b, Codellama-13B, Deepseek-6.7b, Deepseek-1.3B, Toollama, ToolAlpaca. We also explored closed-source LLMs like GPT-4 and GPT-3.5 for closed-source

benchmarking. We also compare the approach with a retriever to reduce the tool selection set.

4.1 Retrievers

Our static API toolset consists of only 9 APIs. Tool retrievers can help decrease the tool descriptions that need to be passed to the LLM, thereby decreasing context length and latency. We compare various retrievers like BM25, FAISS, Ensemble, and DPR. Further, tool retrievers become essential for stability when our pipeline is applied to bigger toolsets. We experiment with multiple tool retrievers, considering each API-argument_name pair as a new tool.

Table 1. Retriever Results

Method	Recall		F1-Score		Time (ms)	
	Top-10	Top-15	Top-10	Top-15	Top-10	Top-15
BM25	0.73	0.83	0.41	0.35	47.8	71.8
FAISS	0.83	0.90	0.46	0.37	758	983
Ensemble (BM25 + FAISS)	0.92	0.95	0.41	0.34	954	1110
Dense Passage Retriever	0.70	0.78	0.40	0.32	—	—

As shown in Table 1, retrievers do not work very well in identifying the correct set of tools, and they also fail to identify the appropriate set of arguments for the tools.

4.2 Closed Source LLMs

Prompting Methods. We experimented with various prompting methods, which include zero-shot prompting, few-shot prompting, chain-of-thought prompting, graph-of-thought prompting, tree-of-thought prompting, and knowledge graph-infused prompting and RTaC prompting, which is our flagship method, in which tools are presented in the form of functions and converted into docstring to be added in the final prompt for solving the query.

Our Experiments on prompting with open-source LLMs like llama2-7b and zephyr-7b-beta showed that both the models hallucinate and perform poorly on our metrics and human evaluation. The JSON similarity score between the actual output and the predicted output was less than 0.1, and the F1-score was less than 0.5 in both cases of zero-shot prompting and few-shot prompting for both the models, which showed that it is not the correct way to evaluate open source models with closed source models.

4.3 Open Source LLMs

We have performed various experiments on tooling LLMs like Toollama, Toolpaca, Codellama, Codegen, and Deepseek. Our Experiments range from few-shot

Table 2. Results - Closed-Source LLMs

Model	JSON Similarity	Precision	Recall	F1-Score
GPT-3.5	67.23	82.34	87.32	84.76
GPT-4 Turbo	74.88	88.23	85.45	86.82
GPT-4 Turbo + SoT	75.23	84.52	89.43	86.91
GPT-4 Turbo + GoT	82.32	87.69	90.16	88.91
GPT-4 Turbo + CoT	79.11	88.12	85.84	86.97
GPT-4 Turbo + ToT	80.69	86.52	88.62	87.56
GPT-4 Turbo + KG	80.62	84.32	90.97	87.52
GPT-4 Turbo + RTaC	87.79	92.12	95.81	93.93

prompting to fine-tuning. We present three pipelines for our Problem Statement that utilize open-source LLMs, incrementally building upon our hypothesis of “Reimagining Tooling as Coding”. This experimentation helps us arrive at our proposed pipeline, RTaC and also serves as an ablation study (Table 2).

Few-Shot prompting of Coding-Base LLMs. Using this pipeline, we investigate our hypothesis around the efficiency of Coding-Base LLMs over normal LLMs. The LLMs are provided with a prompt similar to that referred to in Sect. 4.1 with few-shot examples involving both static tool usage (5 examples) and conditional, iterative tool (3 examples) (Table 3)

Table 3. Results - Few-Shot prompting of Coding-Base LLMs

Dataset	Model	JSON Similarity	Precision	Recall	F1-Score
Static	CodeLlama 7B	68.78	81.25	77.85	79.51
	CodeLlama 13B	73.47	82.80	85.45	86.82
	DeepSeek 1.3B	60.28	73.33	63.00	67.77
	DeepSeek 6.7B	62.02	90.72	61.48	73.29
Dynamic	CodeLlama 7B	67.94	78.24	74.97	76.57
	CodeLlama 13B	71.12	79.84	85.41	82.53
	DeepSeek 1.3B	60.82	65.29	66.53	65.90
	DeepSeek 6.7B	63.96	92.10	57.53	70.82
Bonus	CodeLlama 7B	56.28	76.43	74.17	75.28
	CodeLlama 13B	59.22	78.13	83.91	80.87
	DeepSeek 1.3B	47.32	63.49	62.18	62.83
	DeepSeek 6.7B	54.87	87.56	59.25	70.68

Discussion: Few-shot prompting on pre-trained LLMs like Llama-2 7B and Zephyr 7B (results provided in Table 4) is substantially surpassed by CodeLlama

and DeepSeek, proving our choice for Coding-Base LLMs. However, this pipeline is plagued by a high context length needed to explain the output formats and chatbot behavior, which further leads to higher latencies. Output inspection reveals that while models correctly tend to solve the query, the format is non-convertible, which is necessary for evaluation.

Table 4. Results - Few-Shot prompting of Llama-2 7B and Zephyr 7B

Dataset	Model	JSON Similarity	Precision	Recall	F1-Score
Static	Llama-2 7B Chat	44.57	40.80	41.95	41.37
	Zephyr 7B Chat	50.74	61.58	79.74	69.49
Dynamic	Llama-2 7B Chat	37.94	35.18	38.17	36.61
	Zephyr 7B Chat	41.12	55.39	47.67	51.24

Tool Memorization with “Add Tool” Token. We build upon ToolLlama style fine-tuning of LLMs, which fine-tunes LLMs using query-output pairs to include support for dynamic tooling. To achieve we use the “Added Tools”: token. Dynamic tools provided at runtime are appended in docstring format after this token, while the query follows the “Query”: token in the input prompt. To instill an understanding of our added tokens, we first generate 50 dynamic tools and queries that interface with them using the Self-Instruct [16] methodology. As described above, three hundred such queries-output-toolset tuples, the 100 bonus query-output pairs, and the Stage 1 Dataset are used for instruction fine-tuning over ten epochs.

Table 5. Results - Tool Memorization with “Add Tool” token

Dataset	Model	JSON Similarity	Precision	Recall	F1-Score
Static	CodeLlama 7B	85.89	92.36	94.31	93.32
	ToolApaca	68.45	81.51	73.85	77.49
	ToolLlama	69.57	86.22	78.13	81.98
	DeepSeek 1.3B	84.68	91.94	94.63	93.27
Dynamic	CodeLlama 7B	75.51	85.45	87.51	86.47
	ToolApaca	63.47	75.93	72.27	74.05
	ToolLlama	66.62	78.24	75.11	76.64
	DeepSeek 1.3B	75.83	82.12	81.85	81.98
Bonus	CodeLlama 7B	83.91	87.67	91.12	89.36
	ToolApaca	67.22	80.33	72.91	76.44
	ToolLlama	68.34	84.96	76.19	80.34
	DeepSeek 1.3B	81.35	89.11	90.71	89.90

Discussion: As shown in Table 5, Experimentations with this pipeline reveal training instability. While more extended training makes the model excel in the

static setting, dynamic tool comprehension and usage take a hit. On the other hand, enabling dynamic tooling with controlled training length leads to parameter and data type hallucinations for the memorized static tools. Memorization further limits this pipeline’s ability to modify and delete tools. All this motivates moving to a pipeline with the least tool memorization. Instruction fine-tuning shows promising adherence to a code output format that is convertible to JSON.

RTaC (Our Proposed Pipeline). Here, we build upon the previous pipeline by replacing the “Added Tools”: token with the “Allowed Tools”: token and appending docstrings for all tools, both static and those added dynamically at runtime, after the token. This pipeline avoids tool memorization and instead promotes docstring comprehension.

In the RTaC framework, the “Allowed Tools”: token is used to specify the complete set of tools (static, dynamic, and conditional/iterative) that the LLM can utilize dynamically across different queries. This inclusive approach enables the model to adapt its tool usage flexibly depending on the query context, enhancing its applicability and efficiency. Conversely, the “Added Tools”: token implies a more constrained approach where only dynamic and conditional/iterative tools are specified at runtime, and the static tools are expected to be memorized by the LLM. This distinction between “Allowed” and “Added” fundamentally impacts the model’s performance, with the “Allowed” approach providing a broader, more versatile toolset that does not require the LLM to memorize tools, thereby reducing cognitive load and potentially increasing accuracy.

Table 6. Results - RTaC

Dataset	Model	JSON Similarity	Precision	Recall	F1-Score
Static	DeepSeek 1.3B	87.73	94.38	93.28	93.82
	CodeGen 2B	87.23	81.33	78.43	78.35
	DeepSeek 6.7B	87.79	93.01	95.05	94.01
	CodeLlama 7B	89.91	94.19	94.59	94.38
Dynamic	DeepSeek 1.3B	81.47	90.67	88.88	89.76
	CodeGen 2B	67.43	65.58	65.01	65.29
	DeepSeek 6.7B	82.17	92.03	92.16	92.09
	CodeLlama 7B	85.57	91.11	93.37	92.22
Modified	DeepSeek 1.3B	82.98	91.49	90.14	90.80
	CodeGen 2B	63.84	69.79	60.91	65.04
	DeepSeek 6.7B	87.61	91.18	91.13	91.15
	CodeLlama 7B	86.34	92.77	93.23	92.99
Bonus	DeepSeek 1.3B	83.96	91.47	92.01	91.73
	CodeGen 2B	55.37	69.79	60.91	65.04
	DeepSeek 6.7B	83.17	91.66	93.12	92.38
	CodeLlama 7B	86.92	92.22	94.91	93.54

Discussion: Table 6, shows commendable results on static and dynamic tooling scenarios for RTaC. Further, models under this pipeline gracefully handle the modification and deletion of static tools. This showcases the models’ ability to comprehend the given docstrings. While the context length increases over the previous pipeline due to adding static tool docstring in each prompt, even small models such as DeepSeek 1.3B perform well under this pipeline, leading to minimal latencies. It must also be noted that CodeGen 2B is not a code instruct model, which explains its poor performance.

5 Results

Our experiments show that RTaC is comparable to GPT-4, not only in terms of accuracy but also in terms of cost/query and latency. This shows that tooling can be compared to function calls, which coding-based LLMs can efficiently handle (Table 7).

Table 7. Final Result and Comparison with GPT-4

Metric	GPT-4	RTaC		
		GPT-4	CodeLlama 7B	DeepSeek 1.3B
F-1 Score	86.82	93.93	93.22	93.28
JSON Similarity	74.88	87.79	87.42	85.73
Cost/Query (\$)	0.0341	0.0312	0.0086	0.0060
Latency (s)	7.32	6.88	7.56	5.25
# of Parameters	170B	1760B	7B	1.3B

6 Conclusion

This paper introduces the concept of Reimagining Tooling as Coding (RTaC), a novel approach that reframes tool usage as a coding-related task. Our method involves fine-tuning LLMs on tool descriptions presented in docstring format. The desired output is formatted as variable assignments derived from API calls during training. To achieve this, we employ a unique dual-agent dataset generation method encompassing tool usage in various scenarios, including static, dynamic iterative, and conditional settings. By leveraging Coding-Base LLMs, which are inherently adept at comprehending coding elements, we perform instruction fine-tuning using this specially curated RTaC dataset. This innovative strategy empowers smaller, open-source Coding-Base LLMs, such as DeepSeek 1.3B and CodeLlama 7B, to achieve performance comparable to leading models like GPT-4. This is achieved with significantly reduced computational requirements and faster response times. RTaC presents a groundbreaking approach to tackling tool-usage challenges with LLMs, paving the way for significant advancements in LLM applications.

7 Future Scope

While RTaC demonstrates promising accuracy, there are two critical areas for improvement: mitigating hallucinations and achieving scalability. This section highlights potential avenues for future research that could significantly enhance the framework’s performance.

Query Reformulation Modules: Prior research [8] has established a strong correlation between query quality and model accuracy. This finding aligns with our observations, where queries formulated to reflect the execution sequence achieve near-perfect accuracy precisely. This underscores the need for further exploration into query-optimizing modules. These modules would be designed to reformulate user queries into a format that the model can process efficiently and accurately.

Tool Retrievers: While docstring comprehension leads to high accuracy on a limited set of tools, scalability motivates the addition of tool retrievers to the pipeline. This will empower RTaC to be scaled to massive API sets and outperform current state-of-the-art methods like Gorilla and ToolLLaMA, which rely on tool memorization.

Improvements in Evaluation: The current evaluation metrics in this research domain, such as JSON Similarity and F1-score, fail to evaluate critical aspects such as correctness and optimality reliably. We find that string and AST-based evaluation is not fit for the task of tooling. The same query can often be answered via multiple sequences of tools, yet they are not scored as such during evaluation. To overcome this, we have been working on a bash-based toolset that can act as a deterministic evaluation benchmark for tooling LLMs. Our methodology involves creating an API set that can be mapped to bash operations on directories and files. Models’ outputs can now be deterministically evaluated by state-matching after API call execution (Fig. 6).

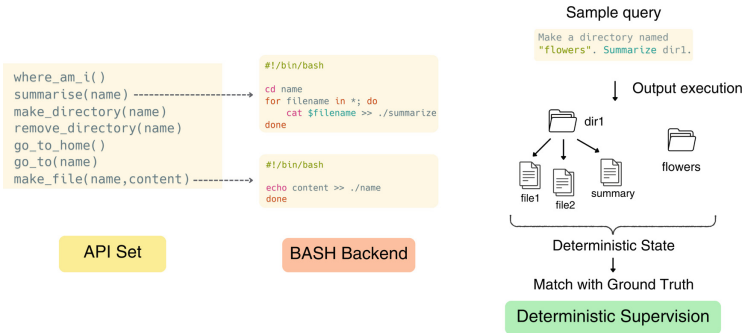


Fig. 6. Design for a bash-based deterministic evaluation benchmark

A Appendix

A.1 JSON Converter

This method is integral to the pipeline, functioning like a compiler. It is designed to transform model-generated code into a specific JSON format. The script categorizes the model's output into two primary types. The first is the General Case, which adheres to a standard variable assignment format using tool names and arguments. The second type is the conditional and iterative case, encompassing additional code structures like conditional statements and for-loops. These structures are used for temporary variable assignments, further expanding the script's capability to handle diverse output formats. In processing these outputs, the script employs several functions. The `process_tool` function is used for the general case, while iterative and conditional cases are managed by specialized bonus handlers that also leverage `process_tool` but with modified parameters. The `make_tool` function checks for the validity of tool and argument names, ignoring invalid entries. The `update_arg_val` function then processes valid arguments. This function is responsible for determining if argument values are lists, handling them recursively if so, and assessing the validity of each value, including scenarios where values are function calls or reference outputs from previous calls, ensuring comprehensive and accurate JSON conversion (Fig. 7).

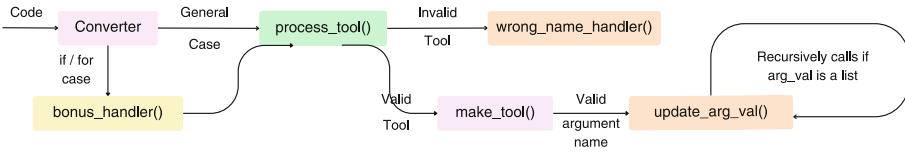


Fig. 7. JSON Conversion Pipeline

A.2 Prompt for Section 4.3

Here are the prompts for Sect. 4.3, testing closed and open-source LLMs for tool usage.

```

The only code you know to write is of type "var_i = function_call (
function_argument)", where i is the ith variable in use. You never output
anything else other than this format. You follow the sequence of completing
the query religiously. You have a given set of functions and you must use
them to answer the query. You are not allowed to use any other functions.

```

```

Here are the allowed functions-
{docstring of the functions}

```

```

Here are some sample queries and their respective responses:
{sample_python}

```

```

Answer very strictly in the same format shown above. Make sure to mention
type argument wherever relevant when calling works_list. Any missing type
arguments is not acceptable. Don't make unnecessary calls to any functions.
When given names, make sure to call search_object_by_name() to get work_ids.
Ensure logical continuity at each step. Ensure that the query is answered
fully. You are not allowed to nest function calls. You are not allowed to
output "python" or any other statement apart from the given format.

```

```
Do not use any other format for output than the one given above. Do not put
any comment in your answer. Anything else other than the format specified is
not acceptable.
Do not define any new helper functions or any other python functions apart
from the ones provided. Do not output any text apart from the final output
code. If you are unable to answer a query, you can output "
Unanswerable_query_error".
```

```
Answer the query: {user query}
```

Listing 1.1. Few Shot Prompting

```
Added Tools: {list of all the dynamic tools}
```

```
Query: {user query}
```

Listing 1.2. Tool Memorization Prompting

```
Allowed Tools: {list of all the tools}
```

```
Query: {user query}
```

Listing 1.3. RTaC Prompting

A.3 Default Tools Used to Generate New Tools

(See Table 8).

Table 8. Default Tools

Tool Description	Functionality
works list	Returns a list of work items matching the request
summarize objects	Summarizes a list of objects. The logic of how to summarize a particular object type is an internal implementation detail
prioritize objects	Returns a list of objects sorted by priority. The logic of what constitutes priority for a given object is an internal implementation detail
add work items to sprint	Adds the given work items to the sprint
get sprint id	Returns the ID of the current sprint
get similar work items	Returns a list of work items that are similar to the given work item
search object by name	Given a search string, returns the id of a matching object in the system of record. If multiple matches are found, it returns the one where the confidence is highest
create actionable tasks from text	Given a text, extracts actionable insights, and creates tasks for them, which are kind of a work item
who am i	Returns the string ID of the current user

References

1. Besta, M., et al.: Graph of thoughts: solving elaborate problems with large language models. arXiv preprint [arXiv:2308.09687](#) (2023)
2. Cai, T., Wang, X., Ma, T., Chen, X., Zhou, D.: Large language models as tool makers. arXiv preprint [arXiv:2305.17126](#) (2023)
3. Hogan, A., et al.: Knowledge graphs. ACM Comput. Surv. (CSUR) **54**(4), 1–37 (2021)
4. Hu, E.J., et al.: Lora: low-rank adaptation of large language models. arXiv preprint [arXiv:2106.09685](#) (2021)
5. Huang, Y., et al.: MetaTool benchmark for large language models: deciding whether to use tools and which to use. arXiv preprint [arXiv:2310.03128](#) (2023)
6. Inaba, T., Kiyomaru, H., Cheng, F., Kurohashi, S.: MultiTool-CoT: GPT-3 can use multiple external tools with chain of thought prompting. arXiv preprint [arXiv:2305.16896](#) (2023)
7. Li, M., et al.: API-bank: a benchmark for tool-augmented LLMs. arXiv preprint [arXiv:2304.08244](#) (2023)
8. Ma, X., Gong, Y., He, P., Zhao, H., Duan, N.: Query rewriting for retrieval-augmented large language models. arXiv preprint [arXiv:2305.14283](#) (2023)
9. Ning, X., Lin, Z., Zhou, Z., Yang, H., Wang, Y.: Skeleton-of-thought: large language models can do parallel decoding. arXiv preprint [arXiv:2307.15337](#) (2023)
10. Parisi, A., Zhao, Y., Fiedel, N.: TALM: tool augmented language models. arXiv preprint [arXiv:2205.12255](#) (2022)
11. Patil, S.G., Zhang, T., Wang, X., Gonzalez, J.E.: Gorilla: large language model connected with massive APIs. arXiv preprint [arXiv:2305.15334](#) (2023)
12. Qin, Y., et al.: ToolLLM: facilitating large language models to master 16000+ real-world APIs. arXiv preprint [arXiv:2307.16789](#) (2023)
13. Schick, T., et al.: ToolFormer: language models can teach themselves to use tools. In: Advances in Neural Information Processing Systems, vol. 36 (2024)
14. Shen, Y., Song, K., Tan, X., Li, D., Lu, W., Zhuang, Y.: HuggingGPT: solving AI tasks with ChatGPT and its friends in hugging face. In: Advances in Neural Information Processing Systems, vol. 36 (2024)
15. Tang, Q., Deng, Z., Lin, H., Han, X., Liang, Q., Sun, L.: ToolAlpaca: generalized tool learning for language models with 3000 simulated cases. arXiv preprint [arXiv:2306.05301](#) (2023)
16. Wang, Y., et al.: Self-instruct: aligning language model with self generated instructions. arXiv preprint [arXiv:2212.10560](#) (2022)
17. Wei, J., et al.: Chain-of-thought prompting elicits reasoning in large language models. In: Advances in Neural Information Processing Systems, vol. 35, pp. 24824–24837 (2022)
18. Yang, K., et al.: If LLM is the wizard, then code is the wand: a survey on how code empowers large language models to serve as intelligent agents. arXiv preprint [arXiv:2401.00812](#) (2024)
19. Yao, S., et al.: Tree of thoughts: deliberate problem solving with large language models. In: Advances in Neural Information Processing Systems, vol. 36 (2024)
20. Ye, J., et al.: ToolEyes: fine-grained evaluation for tool learning capabilities of large language models in real-world scenarios. arXiv preprint [arXiv:2401.00741](#) (2024)
21. Ye, J., et al.: RoTBench: a multi-level benchmark for evaluating the robustness of large language models in tool learning. arXiv preprint [arXiv:2401.08326](#) (2024)

22. Yuan, S., et al.: EASYTOOL: enhancing LLM-based agents with concise tool instruction. arXiv preprint [arXiv:2401.06201](https://arxiv.org/abs/2401.06201) (2024)
23. Zhang, K., Chen, H., Li, L., Wang, W.: Syntax error-free and generalizable tool use for LLMs via finite-state decoding. arXiv preprint [arXiv:2310.07075](https://arxiv.org/abs/2310.07075) (2023)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

