ToolCPT: Improving Tool Utilization in LLM Agents via Continuous Pre-training

Anonymous ACL submission

Abstract

Autonomous agents powered by large language models (LLM-based agents) are capable of using off-the-shelf tools to interact with the environment, solve real-world problems, and boost work efficiency. However, current approaches to enhancing tool use for LLM-based agents primarily focus on post-training fine-tuning or test-time context extension. These methods overlook the fundamental tool knowledge acquisition during the early training phase, where models actually learn and internalize core knowledge representations, restricting model performance on out-of-distribution tool usage. To solve such a problem, we introduce enhancing **tool** knowledge for LLM-based agents during **c**ontinuous **p**re-**t**raining (**ToolCPT**). We identify and bridge a key gap in current LLM training by shifting focus from tool-calling patterns to deep internalization of core tool-knowledge representations. We begin by curating 5.1 million code artifacts from large-scale, high-quality code repositories. These artifacts are selected based on a set of criteria that defines a usable "proxy agent tool", thereby forming a comprehensive agent tool library. For each proxy tool, we then create a detailed playbook covering implementation specifications, core functionalities, interaction protocols with other tools, and illustrative positive and negative examples. This process yields a large-scale tool knowledge corpus comprising 18 billion tokens, which is used to continuously pre-train our model. Experiments show our playbook-enhanced corpus catalyzes deep knowledge internalization, driving the model to notable performance gains on multiple standard benchmarks.

1 Introduction

Recent advances in large language models (LLMs) have boosted the development of autonomous agents capable of using tools and performing reasoning (Li et al., 2025a; Qiu et al., 2025; He et al., 2025; Lu et al., 2025), which are termed LLM-based agents. Currently, the development of such agents mainly follows two paradigms. One paradigm of work focuses on the post-training phase (Liu et al., 2024a; Qian et al., 2025), where LLMs learn from curated tool-use trajectories to solve problems using predefined or previously encountered tools. The second paradigm shifts the focus to the inference stage, optimizing performance by integrating external frameworks and augmenting the context with retrieved knowledge during task execution (Hu et al., 2025; Yan et al., 2025).

Despite their progress, both approaches face limitations. Post-training methods often rely on small-scale, synthetically generated tool sets, which fail to represent the true diversity and distribution of real-world tools. Moreover, agents learn tool knowledge implicitly and incompletely from biased interaction trajectories, where some tools are over-represented while others are entirely absent. On the other hand, methods that enhance context during testing (Hu et al., 2025; Yan et al., 2025) introduce system complexity and potential safety risks through external retrieval and integration mechanisms (Wang et al., 2025).

Although recent efforts (Su et al., 2025; Wu et al., 2025; Zhuang et al., 2025) have attempted to enhance agent capabilities during pre-training, they still lack an effective way to systematically incorporate tool knowledge.We draw inspiration from an import observation that *while tool technology specifically designed for LLM-based agents (Yao et al., 2023) remains in its early stages with a limited set of existing tools, decades of human software development have created a vast ecosystem of code artifacts that serve as a strong proxy for agent tooling. These real-world code artifacts can be adapted and learned from, providing a rich source of prior knowledge for developing capable tool-using agents.* Using this rich repository of real-world code as a proxy offers a powerful way to

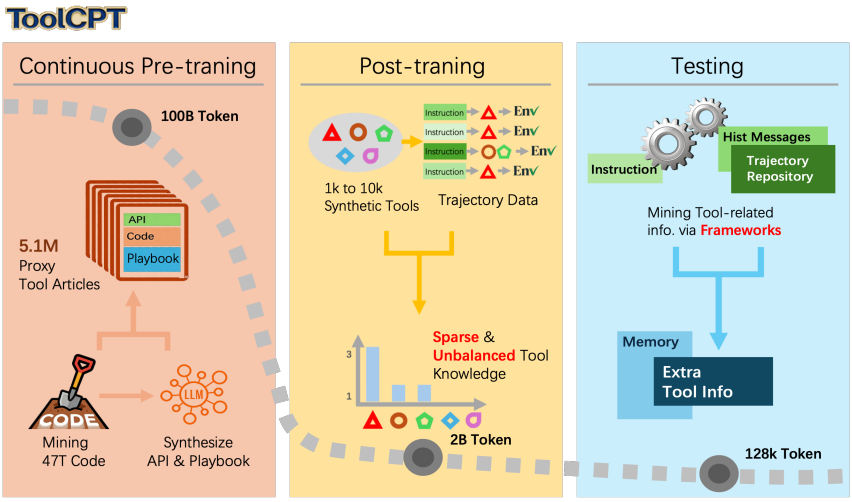

Figure 1: Paradigms for tool learning differ markedly across stages. Post-training relies on limited synthetic tools, resulting in imbalanced knowledge and singular feedback. Testing employs complex frameworks to match tool knowledge, but introduces high complexity and safety risks. The proposed ToolCPT addresses this by mining realistic tool functions from large-scale code, synthesizes interfaces and playbook to construct knowledge, and directly enhances Agent capability during the CPT phase.

bridge the gap between general-purpose LLMs and capable, tool-using agents. This is the core motivation of our work.

In this paper, we propose **ToolCPT**, a method that enhances the tool-use capabilities of LLM-based agents by infusing explicit **tool** knowledge during **c**ontinuous **p**re-**t**raining (CPT). Departing from the limitations of post-training or test-time adaptation, our approach grounds the agent's foundational knowledge in a large-scale, real-world code corpus. As illustrated in Figure 1, we first mine 5.1 million function implementations from 74 trillion tokens of high-quality code that are compatible with agent invocation. These functions are normalized into standardized interfaces, forming what we call **proxy tools**—distinct from agent tools specifically designed for LLM-based agents.

For each proxy tool, we synthesize a comprehensive knowledge playbook covering its implementation specifications, core functionalities, interaction protocols, and illustrative positive/negative usage examples. This process yields a large-scale tool knowledge corpus of 18 billion tokens, which serves as the training data for continuous pre-training. To assess the quality and relevance of the mined proxy tools relative to real-world agent tools, we develop a multi-phase validation framework that combines heuristic rules and open-source LLMs. Rigorous manual annotation of 1,224 Model Context Protocol (MCP) entries confirms a strong alignment between our proxy tools and actual agent tools, as shown in Figure 2.

ToolCPT addresses a key limitation of existing methods, which often treat tools as black-box interfaces and conceal their implementation details from the LLM. This superficial approach restricts learning to pattern matching between user instructions and API signatures, limiting the model's ability to reason about tool functionality or compose tools creatively. In contrast, ToolCPT provides a rich, structured representation of each proxy tool, enabling deeper functional understanding, while deliberate diversification in sequence and style further enhances learning robustness. The resulting knowledge corpus undergoes rigorous multi-expert review to ensure mining rationality, correctness, and readability.

To evaluate the effectiveness of ToolCPT, we design an end-to-end testing scheme. After a standard alignment phase, agents pre-trained with our synthesized corpus are evaluated on multiple benchmarks requiring tool use and environmental interaction. Experimental results demonstrate that agents enhanced with ToolCPT achieve notable and consistent performance gains, validating that grounding continuous pre-training in rich, structured tool knowledge is a powerful and promising direction for building more capable and generalizable LLM-based agents.

The main contributions of this paper are as follows:

- **Large-scale Proxy Tool Extraction:** We

Dataset	Tool Source	SFT Data Scale	Number of Tools	Tool Non-invocation Rate	Maximum Number of Tool Invocations	Average Number of Tool Invocations
ToolACE(Liu et al., 2024a)	Synthesized	21,875	15,962	34.82%	14	1.11
SealTools(Wu et al., 2024)	Synthesized	12,022	3,818	4.16%	36	6.07
ToolAlpaca(Tang et al., 2023)	Synthesized	4,089	1,699	7.06%	393	4.49
APIBank(Li et al., 2023)	Synthesized	16,708	1,768	16.80%	3,673	3.79
APIGen(Liu et al., 2024b)	Synthesized	59,999	3,605	14.40%	921	16.64
Nemotron(Bercovich et al., 2025)	Synthesized	309,275	13,895	9.41%	7,179	37.67
Toucan(Xu et al., 2025)	Collected	118,347	2,200	36.41%	2,927	62.41

Table 1: From a tool perspective, this table analyzes open-source post-training data, including the number of tools and their usage in tool-use trajectories.

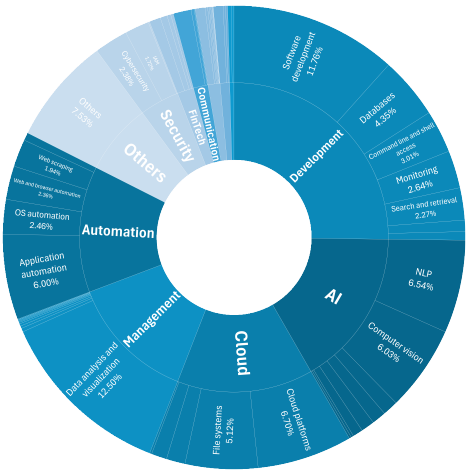

Figure 2: We classify one million proxy tools into all predefined subcategories based on the entry statistics of the MCP. The results show that the tool distribution is generally balanced and fully covers all predefined sub-categories.

mine 5.1 million agent-compatible proxy tools from extensive, high-quality code repositories. Through a multi-phase framework, we ensure the quality and functional relevance of extracted tools, subsequently standardizing their interfaces for agent use.

- **Tool-Knowledge Corpus Construction:** We enrich the mined proxy tools with a comprehensive, multi-perspective playbook—including implementation details, functional specifications, and practical usage examples—to construct a structured, 18-billion-token corpus specifically designed for continuous pre-training. This enables models to develop deep, functional understanding rather than superficial interface matching.

- **Enhanced Agent Capabilities:** We demonstrate that continuous pre-training with our synthesized tool knowledge corpus substantially improves the tool-use capabilities of agents. Rigorous end-to-end evaluation on multiple benchmarks confirms consistent performance gains and enhanced generalization.

2 Related Work

Tool Learning in Pre-training The training of LLM-based Agents suffers from a scarcity of natural Agent data, which complicates data fitting in post-training stages (Chen et al., 2024). Some research address this issue in continuous pre-training phase (Su et al., 2025; Wu et al., 2025), which is closer to post-training. For instance, certain studies incorporate extensive trajectory data into pre-training (Su et al., 2025; Wu et al., 2025) to enhance exploration space or chain-of-thought reasoning capabilities. Beyond using trajectories, other work (Zhuang et al., 2025) also integrates a limited set of real tool playbook that details tool inputs and outputs.

Our work incorporates a wide range of tools in continuous pre-training phase. Moreover, we deliver adequate learning materials for each tool, enabling the LLM to develop a comprehensive understanding of every tool.

Tool Learning in Post-training In post-training, LLMs learn tool knowledge from human-designed trajectories or through online interaction.

Online interaction with tools provides a direct and effective learning approach, typically guided by reward models (Xia et al., 2025). However, this method requires tool authenticity and stability, which restricts current work to a narrow set of tools, such as code and search (Li et al., 2025b; Feng et al., 2025), or to limited sandbox settings (Qian et al., 2025; Singh et al., 2025).

Learning from human-designed trajectories lowers the barrier and scales to larger tool sets (Liu et al., 2024a; Wu et al., 2024; Tang et al., 2023; Li et al., 2023; Liu et al., 2024b; Bercovich et al., 2025; Team et al., 2025; DeepSeek-AI et al., 2025).

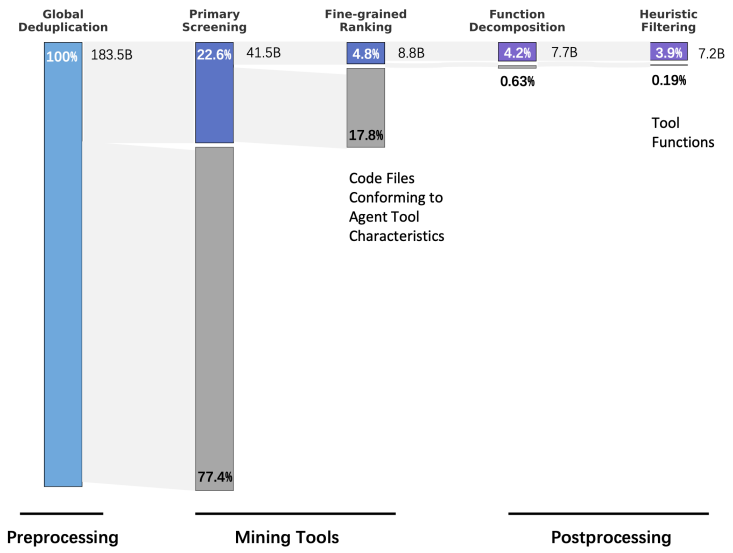

Figure 3: Data Cleaning Pipeline: The process of selecting functions that serve as proxy tools from deduplicated, high-quality code data.

Still, these studies cover around ten thousand tools, and the tools are mostly synthetic, as shown in Table 1. As discussed in experiment section, the distribution of synthetic tools diverges from real tools. As shown in Table1, the trajectory data also shows a strong imbalance: in one dataset (Xu et al., 2025), a tool receives 2,927 calls, while 36.41% tools receive none. The average tool call count is 62.41, but each call provides limited different information.

Tool Learning in Testing Due to the low efficiency of learning tool knowledge during post-training, agents often show limited generalization on out-of-distribution tasks. To address this issue, some studies retrieve both successful and failed trajectories of tool usage during testing (Hu et al., 2025; Yan et al., 2025), employing few-shot strategy to improve invocation accuracy. However, this approach not only increases system complexity, but also introduces greater safety risks (Wang et al., 2025; Dong et al., 2025).

Our work introduces a large number of real-world tools during the continuous pre-training phase and enriches tool-related invocation knowledge. This approach enhances the agent's generalization ability on out-of-distribution tasks.

3 Method

This section outlines the pipeline of ToolCPT. The process begins with the collection of a large-scale corpus of real-world code artifacts, as detailed in Section 3.1. From this candidate pool, we apply a heuristic data cleaning method, described in Section 3.2, to identify and select artifacts suitable for use as proxy tools. These proxy tools are then augmented through a data synthesis pipeline, which consists of knowledge generation, element assembly, and quality filtering steps, as presented in Section 3.3. Finally, the mined proxy tools and their synthesized knowledge are combined and formatted into a continuous pre-training dataset using a templating approach, which is explained in Section 3.4.

3.1 Data Collection

To collect data related to tool usage, we gathered 47 trillion tokens of code data, comprising 140 billion files. To better approximate how agent tools operate, we focused on self-contained code by filtering out files that rely on external dependencies. Additionally, given that 97% of MCP tools are implemented in four programming languages—Python, Go, JavaScript, and TypeScript, we restricted our dataset to code written in these languages. For quality control, we only included Git repositories with more than zero stars. After this filtering process, we obtained 1.3 trillion tokens of data. Following file-level deduplication, we arrived at 183.5 billion tokens of code data. This dataset supported further semantic filtering.

We also collected a substantial set of real-world MCP tools for validation. In Section 4.1, we use these real tools to verify the authenticity of proxy tools. Sourced from 2,123 Git repositories hosting MCP implementations, the data is filtered

to include only the four programming languages previously mentioned. Tool lists are extracted from these MCPs via API calls. To capture implementation insights, we then locate and retrieve the relevant source code segments for each tool from the corresponding repositories. Following manual annotation, we obtain 1,224 distinct tool lists, covering a total of 10,901 individual agent tools.

3.2 Proxy Tool Mining

Although the collected artifacts inherit most characteristics of agent tools, important differences exist. Many code functions only perform simple internal operations—they lack complexity and have little value for independent user invocation. Therefore, we apply LLM-based mining and heuristic rules to the filtered code files. The data flow is shown in Figure 3. We define six key properties of agent tools: 1) a clear calling interface, 2) the ability to obtain external information, 3) the ability to change the environment, 4) no need for human intervention during execution, 5) clear feedback, and 6) usage scenarios beyond low-level implementation details. Using an open-source LLM, we score tools based on these properties and format the output. We show an example in Table 2.

Criterion	Score	Explanation
Clear Interface	✓	It has interface.
Acquire External Info	✓	It calls the Spotify API to obtain a user's recently played songs.
Change Environment	✓	It saves the data to a local JSON file.
No Intervention Needed	✓	The code executes independently.
Clear Feedback	✓	It returns outputs or error messages.
Use Scenarios	✓	It applies to the popular entertainment domain.

Table 2: LLM scoring and assessment of **get_recently_played_songs** across six criteria.

Agent tools usually exist as tool lists, with dependencies or collaborations among other tools. Thus, we assign scores at the level of entire code files rather than individual functions.

To mine the proxy tools more objectively, we employ multiple series of open-source LLMs to score code files. We first compare human annotations with model scores on a validation set of 100 files. After evaluating multiple models, we select the Qwen (Qwen et al., 2025) and GPT-OSS (OpenAI, 2025) series to streamline the screening process with reduced bias. A multi-stage framework is adopted to balances speed and accuracy. The primary screening stage uses the Qwen3-3B model for initial scoring. To ensure high recall, we retain artifacts that meet at least three of the six defined agent tool features. During fine-grained ranking stage we apply the GPT-OSS-120B model with the same instructions to rescore the initial results. Files meeting at least four features are kept.

We then post-process the mined artifacts. Code files are decomposed into separate tool functions, and helper code segments are removed. Heuristic rules further filter the functions: for example, we discard tool lists with over 25 tools, remove homogeneous functions like main, __init__, and get_args, and filter out 50% of tools with no input parameters. Finally, this process yields 5.1 million proxy tools with their implementation code.

We further generate a tool-calling interface for each tool function. These interfaces follow the MCP-defined tool format, with details on tool capabilities, parameter descriptions, and requirements. We also randomly include real tool descriptions as few-shot prompts to guide the model to produce more diverse tool types.

3.3 Tool Playbook Generation

The interaction between an agent and the real world through tools is a dynamic process. Learning only static knowledge, such as tool interfaces and corresponding code, does not effectively enhance the tool-calling capability of LLMs. We address this by equipping each tool with a playbook. The playbook uses a structured data format to describe potential dynamic issues in the tool-calling process from 5 aspects and 12 perspectives. We present a schematic of the playbook and its organization with code and API in Figure 7.

Basic Information This section extends the tool-calling interface. It describes the function of the tool, the parameters and the return format. *A key problem is that agents, lacking tool-specific knowledge, frequently disregard parameter constraints, leading to parameter hallucination. Explicitly stating the parameter specification rules here tends to mitigate this issue.*

Background Knowledge This section introduces tool-related knowledge from several perspectives. It first covers the application scenarios of the tool and the relevant terminology, specifying when the tool is appropriate or inappropriate. Then it explains the implementations and data flow of the tool with reference to its code. Finally, if multiple

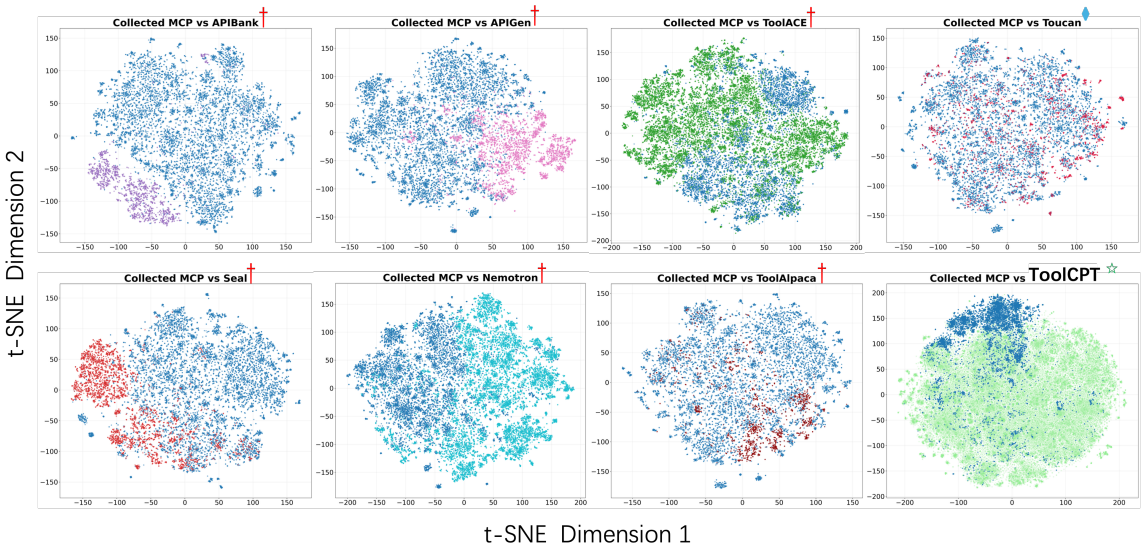

Figure 4: Feature Space Distribution: The distribution of tool descriptions for proxy tools, synthetic tools, and real tools in the feature space. Blue points in each subfigure denote real tools. In each title of subfigure, ⋆ represents proxy tools, † represents synthetic tool, while ⧫ represents real tools.

related tools exist in the tool list, it introduces the remaining tools.

Calling Methods Tool calling methods are essential. We provide at least three correct and three incorrect calling examples for each tool. The model learns the calling method through comparison and explanation.

Tool Coordination Complex tasks require multiple tools to work together. When the tool list includes multiple tools, this section explains tool coordination. First, we statically analyze relationships between tools, such as collaboration, mutual exclusion, or independence. Then, we list all reasonable tool combinations. These include different calling sequences, multiple calls of a single tool, and their combinations. Unlike planning from complex tasks, we start from tool combinations to discuss achievable tasks.

Error Analysis Tool feedback is key for agent-environment interaction, but is often overlooked. *Most trajectory data contain synthetic feedback with only correct responses. This section analyzes error feedback. Error feedback includes two types: feedback from incorrect calls and feedback from environment errors.*

Playbook Generation Strategy We use LLMs to generate functional descriptions of tools. The input includes the tool's interface and implementation code; if adjacent tools exist in the tool list, they are also included. After evaluating several open-source LLMs based on metrics such as information completeness, hallucination level, and output diversity, we employ GPT-OSS-120B for playbook generation.

3.4 Templating Proxy Tool Knowledge

After obtaining the tool interface, implementations, and relevant playbook, we convert these elements into pre-training documents through rule-based concatenation. The concatenation methods follow two design principles: first, the presentation format resembles a science article about tool knowledge; second, the Chinese-to-English content ratio is 1:2. Moreover, since rearranging the three elements yields information gain, we design ten permutation sequences for them, including three binary combinations and seven ternary combinations. To enhance the model's understanding through order comparison, 30% of the data incorporate two different permutation sequences. Based on this procedure, we obtain the final proxy tool knowledge-enhanced synthetic data. A complete example can be seen in Appendix B.

3.5 Data Generation Overhead

Thirty professional code annotators debug and filter 2,123 real MCP cases and extract corresponding tool codes over 20 days. A heuristic algorithm first pre-processes 47T of code data using 7 CPU-days. 350K GPU-hours are dedicated to tool min-

ing. Subsequently, 1.3K GPU-hours generate interfaces and playbook for each tool. To ensure data quality, nine LLM experts perform multiple rounds of sampling inspection. Detected issues help establish heuristic cleaning rules and an LLM-based quality scoring filter. The process yields enhanced synthetic data for agent tool knowledge.

4 Experiment

This study evaluates proxy tools and synthesized pre-training data via two experiments. Section 4.1 examines the similarity between mined proxy tools and real tools in spatial and label distributions. Section 4.2 uses end-to-end evaluation, post-training after continuous pre-training, to measure synthesized data's impact on agent performance.

4.1 Similarity Between Proxy and Real Tools

We use 10k annotated MCP tools as reference and evaluate mined proxy tools' effectiveness via spatial and label distributions similarity. Open-source synthetic tools (Liu et al., 2024a; Wu et al., 2024; Tang et al., 2023; Li et al., 2023; Liu et al., 2024b; Bercovich et al., 2025; Team et al., 2025; DeepSeek-AI et al., 2025) and real tools (Xu et al., 2025) are included for comparison.

We first extract all tools' descriptions, convert to uniform language, encode with bge-m3 model (Chen et al., 2025b), and visualize tool feature space using t-SNE (Maaten and Hinton, 2008). For clarity, 100k proxy tools are randomly selected, while all open-source tools are fully included. Results are shown in Figure 4. Tools with distributions well blending with MCP tools are considered closer to real tools. The figure shows synthetic tools in the first two columns have distinct distributions from real tools. The third column performs better but remains separable. Toucan achieves the best alignment as its tools are also collected from MCPs. Proxy tools in ToolCPT completely overlap with real tools' region and supplies tools in sparse areas, qualitatively demonstrating proxy tools' authenticity and supplement to limited real tools.

We further quantify tool-type distribution similarity. Following Kimi k2 (Team, 2025) and MCP markets' categorization (mcp, a,c; cur; mcp, b; byt; cli; ali; cla; dif) , tools are divided into 12 main categories and 58 sub-categories, see Table 7. Using an LLM, we collect category distributions for 10k real tools and 1M mined tools, presented in Figure 5 and Figure 6. We further quantify the cosine similarity between the two tool types based on their label distributions. Cosine similarity between the two distributions is 94.5% at main-category level and 70.1% at sub-category level.

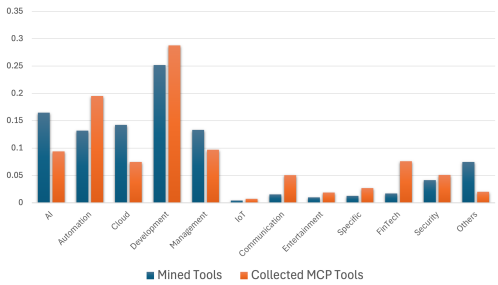

Figure 5: Tool Tag Distribution: The distribution of proxy tools and real tools across tool tags.

We report the distribution similarity between other open-source and real tools in Appendix A.3. Proxy tools show notable similar distributions compared to synthetic ones. We also investigate in depth why synthetic tools deviate from real ones.

4.2 End-to-End Evaluation

Agents rely on strong instruction-following capability for planning and execution during interactions, making their ability difficult to assess directly in the pre-training stage. We design an end-to-end protocol: after continuous pre-training, the model undergoes lightweight instruction alignment, and its performance is then evaluated on open-source agent benchmarks.

Experimental Setup Base models are Qwen2.5-7B-Base (Qwen et al., 2025) and Llama3.1-8B-Base (Grattafiori et al., 2024). Continuous pre-training uses 100B tokens from Dolma 3 Dolmino Mix (AllenAI, 2025) as base corpus, with equal token amounts replaced by the proxy tool data. A multi-stage uniform sampling strategy reduces variance (e.g., 5% replacement uses 15% general data from the replaced portion of 20% replacement runs).

Post-Training Lightweight instruction alignment uses 1.7B tokens of in-house general instruction data. To minimize the impact of post-training, agent-related instructions constitute only 7.2% of this data, with no overlap against the test sets. Moreover, an augmented variant adds 7,000 agent data samples from seven public instruction sets, mentioned in Table 1, for comparison.

Evaluation Benchmarks We employ two groups of benchmarks for evaluation: (1) 8 agent-related

Model	Ratio of Tool Knowledge	Browse Comp-ZH	Browse Comp	GAIA v2	xBench v2	ACE Bench	TauBench v1	TauBench v2	BFCL v3	Avg.
Qwen2.5-7B	0%	4.17%	2.36%	25.00%	26.26%	24.00%	24.24%	20.00%	10.25%	17.04%
	0%⋆	8.33%	1.57%	31.03%	23.47%	30.00%	21.21%	18.05%	16.13%	18.72%
	5%	9.72%	3.94%	31.59%	33.00%	40.00%	30.91%	26.98%	13.50%	23.71%
	10%	9.03%	3.94%	30.68%	37.00%	43.00%	27.88%	27.00%	15.00%	**24.19%**
	20%	12.50%	5.51%	30.68%	34.00%	42.00%	29.09%	19.13%	10.80%	22.96%
Llama3.1-8B	0%	5.56%	3.94%	25.00%	25.00%	30.00%	29.09%	20.86%	12.64%	19.01%
	10%	9.72%	5.51%	32.95%	30.00%	44.00%	36.36%	24.46%	14.52%	**24.69%**

Table 3: Model Performance: The performance of Qwen and Llama base models on agent-related benchmarks after pre-training with different ratios of synthetic data and subsequent simple instruction alignment. The symbol ⋆ indicates agent-specific data augmentation during post-training.

Model	with Code	with API & Playbook	with Multi-Template	Browse Comp-ZH	Browse Comp	GAIA v2	xBench v2	ACEBench	TauBench v1	TauBench v2	BFCL v3	Avg.
Qwen2.5-7B	✓			8.05%	2.52%	31.59%	29.00%	31.60%	27.88%	21.80%	12.53%	20.62%
	✓	✓		9.03%	3.94%	30.68%	37.00%	36.00%	27.88%	21.94%	9.49%	22.00%
	✓	✓	✓	9.72%	3.94%	31.59%	33.00%	40.00%	30.91%	26.98%	13.50%	**23.71%**

Table 4: The contribution of different components within synthetic data to accuracy improvement.

Model	Ratio	MMLU	GSM8k	BBH	SQA	MBPP+	Avg.
Qwen2.5-7B	0%	80.41%	94.40%	83.89%	6.58%	74.31%	67.92%
	10%	80.94%	94.60%	83.43%	7.12%	74.60%	68.14%
Llama3.1-8B	0%	78.40%	92.85%	84.45%	8.77%	66.33%	66.16%
	10%	78.57%	91.29%	83.76%	8.87%	66.97%	65.89%

Table 5: The impact of training with synthetic data on the model's general abilities.

tasks, with 4 AI search: BrowseComp-ZH (Zhou et al., 2025), BrowseComp (Wei et al., 2025), GAIA v2 (Russell et al., 2025), xBench v2 (Chen et al., 2025c) and 4 common tool-calling: ACEBench (Chen et al., 2025a), TauBench v1 (Yao et al., 2024), TauBench v2 (Barres et al., 2025), BFCL v3 (Charlie Cheng-Jie Ji); (2) 5 general benchmarks related to knowledge, reasoning, mathematics and coding: MMLU (Hendrycks et al., 2021), GSM8k (Cobbe et al., 2021), BBH (Suzgun et al., 2022), SimpleQA (OpenAI, 2024), MBPP+ (Liu et al., 2023). Pass@5 average performance is reported to reduce variance.

Main Results Table 3 presents agent-related benchmark results under different synthetic data mixtures. Introducing 10% tool knowledge data improves the average performance of Qwen 7B and Llama 8B by 7.15% and 5.68%, respectively. This confirms our synthetic data effectively enhances the agent's out-of-distribution tool-use ability. However, increasing the mixture ratio to 5%, 10%, and 20% does not yield linear performance gains. The 20% ratio causes a performance drop, suggesting excessive single-task data dilutes training signals from other data. We also find post-training with more data yields smaller gains than pre-training with incorporated data, as demonstrated by the sub-optimal performance of the second row (0%*) in Table 3.

Ablation Study Table 4 presents the results of an ablation study using a 5% data mixture. The ablation tests the contribution of each stage: pre-training solely on mined proxy tools (Section 3.2); training on proxy tools augmented with synthesized knowledge (Section 3.3); and finally, training on the complete ToolCPT dataset, which applies templating to integrate proxy tools and their corresponding knowledge (Section 3.4). The results show that both the synthesized tool knowledge and the template designs—including language and structural ordering—contribute to performance improvements.

Analysis on general benchmarks in Table 5 shows synthetic data does not harm the model's other capabilities. This is because the synthetic data contains code and high-quality knowledge, maintaining a quality level comparable to that of general pre-training data.

5 Conclusion

To address tool-using LLM agents' insufficient tool knowledge, we propose ToolCPT. This code-grounded paradigm mines proxy tools, synthesizes a tool knowledge corpus, and conducts continuous pre-training to enhance tool understanding. Evaluations confirm ToolCPT improves cross-benchmark performance, verifying pre-training with explicit tool knowledge builds robust agents.

Limitations

This study constructs a proxy tool based on code data mining and develops corresponding tool knowledge playbook according to the implemented code and associated functions. The playbook includes possible feedback and error reports for single-tool execution, as well as examples of multi-tool collaboration. However, these proxy tools remain static simulations without actual execution. The immediate feedback and error descriptions are inferred reasonably from code analysis, but cannot capture the state changes of the environment during continuous tool calls. Although multi-tool coordination and sequential multi-task execution are important capabilities of agents, this work does not address these issues. Further research will explore them in greater depth.

References

Ali mcp market. https://bailian.console.aliyun.com/?tab=mcp#/mcp-market. Accessed: 2026-01-04.

Bytedance mcp marketplace. https://www.volcengine.com/mcp-marketplace?utm_source=ai-bot.cn. Accessed: 2026-01-04.

Claude mcp server. https://www.claudemcp.com/servers. Accessed: 2026-01-04.

Cline mcp server. https://cline.bot/mcp-marketplace. Accessed: 2026-01-04.

Cursor servers. https://cursor.directory/mcp. Accessed: 2026-01-04.

Dify. dify.ai. Accessed: 2026-01-04.

a. Mcp market. https://mcpmarket.com/. Accessed: 2026-01-04.

b. Mcp servers. https://mcpservers.org/. Accessed: 2026-01-04.

c. mcp.so. https://mcp.so/. Accessed: 2026-01-04.

AllenAI. 2025. Dolma 3 dolmino mix (100b). https://huggingface.co/datasets/allenai/dolma3_dolmino_mix-100B-1025. Accessed: 2026-01-01.

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. 2025. tau2-bench: Evaluating conversational agents in a dual-control environment. *arXiv preprint arXiv:2506.07982*.

Akhiad Bercovich, Itay Levy, Izik Golan, Mohammad Dabbah, Ran El-Yaniv, Omri Puny, Ido Galil, Zach Moshe, Tomer Ronen, Najeeb Nabwani, Ido Shahaf, Oren Tropp, Ehud Karpas, Ran Zilberstein, Jiaqi Zeng, Soumye Singhal, Alexander Bukharin, Yian Zhang, Tugrul Konuk, and 114 others. 2025. Llama-nemotron: Efficient reasoning models. *Preprint*, arXiv:2505.00949.

Fanjia Yan Shishir G. Patil Tianjun Zhang Ion Stoica Joseph E. Gonzalez Charlie Cheng-Jie Ji, Huanzhi Mao. Gorilla bfvl v3. https://gorilla.cs.berkeley.edu/leaderboard.html. Accessed: 2025-01-17.

Chen Chen, Xinlong Hao, Weiwen Liu, Xu Huang, Xingshan Zeng, Shuai Yu, Dexun Li, Shuai Wang, Weinan Gan, Yuefeng Huang, and 1 others. 2025a. Acebench: Who wins the match point in tool usage? *arXiv preprint arXiv:2501.12851*.

Jianlv Chen, Shitao Xiao, Peitian Zhang, Kun Luo, Defu Lian, and Zheng Liu. 2025b. M3-embedding: Multi-linguality, multi-functionality, multi-granularity text embeddings through self-knowledge distillation. *Preprint*, arXiv:2402.03216.

Kaiyuan Chen, Yixin Ren, Yang Liu, Xiaobo Hu, Haotong Tian, Tianbao Xie, Fangfu Liu, Haoye Zhang, Hongzhang Liu, Yuan Gong, and 1 others. 2025c. xbench: Tracking agents productivity scaling with profession-aligned real-world evaluations. *arXiv preprint arXiv:2506.13651*.

Zehui Chen, Kuikun Liu, Qiuchen Wang, Wenwei Zhang, Jiangning Liu, Dahua Lin, Kai Chen, and Feng Zhao. 2024. Agent-flan: Designing data and methods of effective agent tuning for large language models. *arXiv preprint arXiv:2403.12881*.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

DeepSeek-AI, Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenhao Xu, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, and 245 others. 2025. Deepseek-v3.2: Pushing the frontier of open large language models. *Preprint*, arXiv:2512.02556.

Shen Dong, Shaochen Xu, Pengfei He, Yige Li, Jiliang Tang, Tianming Liu, Hui Liu, and Zhen Xiang. 2025. Memory injection attacks on llm agents via query-only interaction. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

Jiazhan Feng, Shijue Huang, Xingwei Qu, Ge Zhang, Yujia Qin, Baoquan Zhong, Chengquan Jiang, Jinxin Chi, and Wanjun Zhong. 2025. Retool:

Reinforcement learning for strategic tool use in llms. *arXiv preprint arXiv:2504.11536*.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, and 542 others. 2024. The llama 3 herd of models. *Preprint*, arXiv:2407.21783.

Yichen He, Guanhua Huang, Peiyuan Feng, Yuan Lin, Yuchen Zhang, Hang Li, and 1 others. 2025. Pasa: An llm agent for comprehensive academic paper search. *arXiv preprint arXiv:2501.10120*.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2021. Measuring massive multitask language understanding. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

Mengkang Hu, Tianxing Chen, Qiguang Chen, Yao Mu, Wenqi Shao, and Ping Luo. 2025. Hiagent: Hierarchical working memory management for solving long-horizon agent tasks with large language model. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 32779–32798.

Bingxuan Li, Yiwei Wang, Jiuxiang Gu, Kai-Wei Chang, and Nanyun Peng. 2025a. Metal: A multi-agent framework for chart generation with test-time scaling. *arXiv preprint arXiv:2502.17651*.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. 2023. Api-bank: A comprehensive benchmark for tool-augmented llms. *arXiv preprint arXiv:2304.08244*.

Xuefeng Li, Haoyang Zou, and Pengfei Liu. 2025b. Torl: Scaling tool-integrated rl. *arXiv preprint arXiv:2503.23383*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In *NeurIPS 2023 Datasets and Benchmarks Track*.

Weiwen Liu, Xu Huang, Xingshan Zeng, Xinlong Hao, Shuai Yu, Dexun Li, Shuai Wang, Weinan Gan, Zhengying Liu, Yuanqing Yu, and 1 others. 2024a. Toolace: Winning the points of llm function calling. *arXiv preprint arXiv:2409.00920*.

Zuxin Liu, Thai Hoang, Jianguo Zhang, Ming Zhu, Tian Lan, Juntao Tan, Weiran Yao, Zhiwei Liu, Yihao Feng, Rithesh RN, and 1 others. 2024b. Apigen: Automated pipeline for generating verifiable and diverse function-calling datasets. *Advances in Neural Information Processing Systems*, 37:54463–54482.

Junting Lu, Zhiyang Zhang, Fangkai Yang, Jue Zhang, Lu Wang, Chao Du, Qingwei Lin, Saravan Rajmohan, Dongmei Zhang, and Qi Zhang. 2025. Axis: Efficient human-agent-computer interaction with api-first llm-based agents. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7711–7743.

Laurens van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605.

OpenAI. 2024. Measuring short-form factuality in large language models. Technical report, OpenAI.

OpenAI. 2025. Introducing gpt-oss. https://openai.com/index/introducing-gpt-oss/.

Cheng Qian, Emre Can Acikgoz, Qi He, Hongru Wang, Xiusi Chen, Dilek Hakkani-Tür, Gokhan Tur, and Heng Ji. 2025. Toolrl: Reward is all tool learning needs. *arXiv preprint arXiv:2504.13958*.

Rui Qiu, Shijie Chen, Yu Su, Po-Yin Yen, and Han Wei Shen. 2025. Completing a systematic review in hours instead of months with interactive ai agents. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 31559–31593.

Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, and 25 others. 2025. Qwen2.5 technical report. *Preprint*, arXiv:2412.15115.

Lloyd Russell, Anthony Hu, Lorenzo Bertoni, George Fedoseev, Jamie Shotton, Elahe Arani, and Gianluca Corrado. 2025. Gaia-2: A controllable multi-view generative world model for autonomous driving. *arXiv preprint arXiv:2503.20523*.

Joykirat Singh, Raghav Magazine, Yash Pandya, and Akshay Nambi. 2025. Agentic reasoning and tool integration for llms via reinforcement learning. *URL https://arxiv. org/abs/2505.01441*, 5.

Liangcai Su, Zhen Zhang, Guangyu Li, Zhuo Chen, Chenxi Wang, Maojia Song, Xinyu Wang, Kuan Li, Jialong Wu, Xuanzhong Chen, and 1 others. 2025. Scaling agents via continual pre-training. *arXiv preprint arXiv:2509.13310*.

Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc V. Le, Ed H. Chi, Denny Zhou, and Jason Wei. 2022. Challenging big-bench tasks and whether chain-of-thought can solve them. *Preprint*, arXiv:2210.09261.

Qiaoyu Tang, Ziliang Deng, Hongyu Lin, Xianpei Han, Qiao Liang, Boxi Cao, and Le Sun. 2023. Toolalpaca: Generalized tool learning for language models with 3000 simulated cases. *arXiv preprint arXiv:2306.05301*.

Kimi Team, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, and 1 others. 2025. Kimi k2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*.

Moonshot AI Team. 2025. Kimi k2: Scaling agentic intelligence with trillion-parameter mixture-of-experts. *arXiv preprint arXiv:2507.12966*.

Bo Wang, Weiyi He, Shenglai Zeng, Zhen Xiang, Yue Xing, Jiliang Tang, and Pengfei He. 2025. Unveiling privacy risks in llm agent memory. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 25241–25260.

Jason Wei, Zhiqing Sun, Spencer Papay, Scott McKinney, Jeffrey Han, Isa Fulford, Hyung Won Chung, Alex Tachard Passos, William Fedus, and Amelia Glaese. 2025. Browsecomp: A simple yet challenging benchmark for browsing agents. *arXiv preprint arXiv:2504.12516*.

Mengsong Wu, Tong Zhu, Han Han, Chuanyuan Tan, Xiang Zhang, and Wenliang Chen. 2024. Seal-tools: Self-instruct tool learning dataset for agent tuning and detailed benchmark. In *CCF International Conference on Natural Language Processing and Chinese Computing*, pages 372–384. Springer.

Weiqi Wu, Xin Guan, Shen Huang, Yong Jiang, Pengjun Xie, Fei Huang, Jiuxin Cao, Hai Zhao, and Jingren Zhou. 2025. Masksearch: A universal pre-training framework to enhance agentic search capability. *arXiv preprint arXiv:2505.20285*.

Yu Xia, Jingru Fan, Weize Chen, Siyu Yan, Xin Cong, Zhong Zhang, Yaxi Lu, Yankai Lin, Zhiyuan Liu, and Maosong Sun. 2025. Agentrm: Enhancing agent generalization with reward modeling. *arXiv preprint arXiv:2502.18407*.

Zhangchen Xu, Adriana Meza Soria, Shawn Tan, Anurag Roy, Ashish Sunil Agrawal, Radha Poovendran, and Rameswar Panda. 2025. Toucan: Synthesizing 1.5 m tool-agentic data from real-world mcp environments. *arXiv preprint arXiv:2510.01179*.

Cilin Yan, Jingyun Wang, Lin Zhang, Ruihui Zhao, Xiaopu Wu, Kai Xiong, Qingsong Liu, Guoliang Kang, and Yangyang Kang. 2025. Efficient and accurate prompt optimization: the benefit of memory in exemplar-guided reflection. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 753–779.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. 2024. tau-bench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. *Preprint*, arXiv:2210.03629.

Peilin Zhou, Bruce Leon, Xiang Ying, Can Zhang, Yifan Shao, Qichen Ye, Dading Chong, Zhiling Jin, Chenxuan Xie, Meng Cao, and 1 others. 2025. Browsecomp-zh: Benchmarking web browsing ability of large language models in chinese. *arXiv preprint arXiv:2504.19314*.

Yuchen Zhuang, Jingfeng Yang, Haoming Jiang, Xin Liu, Kewei Cheng, Sanket Lokegaonkar, Yifan Gao, Qing Ping, Tianyi Liu, Binxuan Huang, and 1 others. 2025. Hephaestus: Improving fundamental agent capabilities of large language models through continual pre-training. *arXiv preprint arXiv:2502.06589*.

A Appendix

The appendix provides additional experimental details and pretraining data. There is more pretraining data in the attachment file. Upon acceptance of the paper, we will release the mining framework and associated data.

A.1 Experiments Details

In this subsection, we present more details regarding the training parameters in Table 6.

Stage	batch_size	warmup_ratio	lr	lr_decay_ratio	seq_len	optimizer
CPT	2048	0.10	2.0×10^{-5}	0.01	32 768	adamw
SFT	128	0.08	2.0×10^{-5}	0.01	32 768	adamw

Table 6: Training Hyperparameters Configuration.

A.2 Distribution of Proxy Tools

This section analyzes the distribution of proxy tools within the tool-type taxonomy. We randomly sample one million proxy tools and label them using GPT-oss-120B. Table 7 shows the tool distribution and specific label statistics. The results indicate that the random sampling covers all long-tail tool types with a balanced distribution.

A.3 Comparison of Proxy, Synthetic, and Real Tools

We further compare the distributions of proxy tools and real tools at the second-level labels, as shown in Figure 6. Because real tools focus on management tasks, proxy tools show small differences on a few labels, but the overall distributions remain highly similar.

We also compute the cosine similarity between tools from open-source agent training datasets and real tools in the label taxonomy, shown in Table 7. Among them, the Toucan dataset—collected from real tools—exhibits higher similarity to real tools than proxy tools. Other synthetic tools show noticeably lower similarity compared to proxy tools.

We conduct a detailed analysis of APIBench, which has the lowest similarity. Synthetic tools show strong homogeneity in tool names and tool functions.

Regarding names, 95.1% of synthetic tool names follow a two-word "verb_noun" pattern, while most remaining names use a "noun_verb" pattern, such as "get_activities" and "medication_check." Tools starting with "Get_xx" alone account for 38.7%. This naming pattern hinders

Table 7: Distribution of Proxy Tools: The distribution of one million proxy tools within a two-level tool tag taxonomy.

Primary Category	Secondary Category	Count
AI	AIGC	9856
	Computer vision	59,719
	NLP	64,813
	Multimodal	1723
	RAG systems	1507
	Speech and audio processing	10,754
	Knowledge and memory	9476
	Embeddings and vector databases	7070
Automation	Application automation	59,389
	Web and browser automation	23,421
	OS automation	24,362
	Robot control	5440
	Web scraping	19,228
Cloud	Cloud platforms	66,402
	Storage and CDN	13,111
	Virtualization and containers	10,820
	Desktop and remote systems	1423
	File systems	50,750
Development	Software development	116,508
	Package management	5967
	Monitoring	26,161
	Command line and shell access	29,783
	Version control and repos	7919
	Search and retrieval	22,484
	Databases	43,124
Management	Data analysis and visualization	123,810
	Business intelligence	1699
	Customer data platforms	696
	Calendar and schedule	1652
	Customer support	505
	Marketing and advertising	725
	Legal compliance	527
	Note taking	1038
	Location services	2551
IoT	Smart home automation	1904
	Manufacturing industrial iot	2468
Communication	Messaging and collaboration	12,495
	Social media	2158
	Mobile Devices	641
Entertainment	Art and culture	1447
	Entertainment and media	1652
	Games and gamification	6942
Specific	Agriculture and environmental	309
	Ecommerce and retail	1813
	Education and elearning	6943
	Health and wellness	1158
	Real estate property	101
	Travel and transportation	801
	Logistics and supply chain	417
	Weather services	1023
FinTech	Blockchain	7894
	Cryptocurrency	2607
	Financial trading	4889
	Payment and settlement	1886
Security	Cybersecurity	23,565
	IAM	17,002
	Data privacy and protection	918
Others	Others	74,551

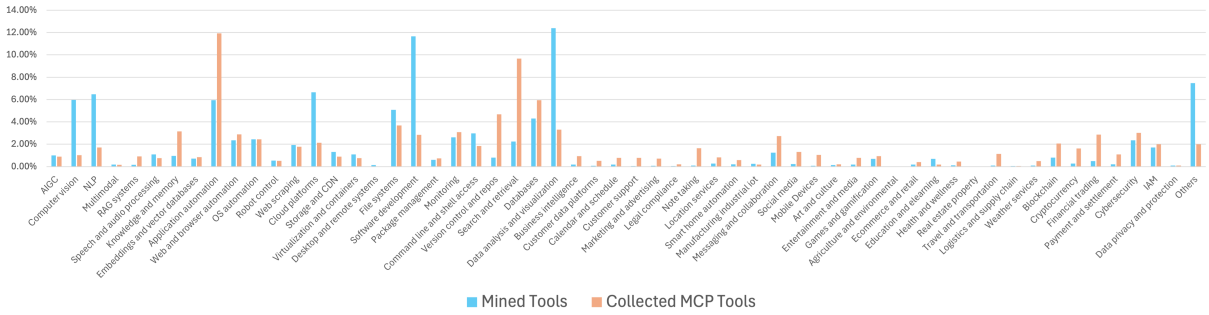

Figure 6: Distribution Across Secondary Tags: The distribution of proxy tools and real tools across secondary tool tags.

Model	Tool Source	Cosine Similarity (%)
ToolCPT	Proxy	94.5
Toucan	Collected	96.2
ToolAlpaca	Synthesized	80.7
API-Gen	Synthesized	78.2
ToolACE	Synthesized	73.6
Nemotron Agent	Synthesized	72.8
SealTools	Synthesized	65.9
APIBank	Synthesized	28.6

Table 8: The cosine similarity of the distribution in the first-level label system between the tools in the open-source dataset and the real MCP tools we collected.

agents in calling out-of-distribution tools. In contrast, proxy tools use real tool names and avoid this issue.

Regarding functionality, synthetic tools are not decomposed based on concrete implementations, but are mapped from specific user-task data, as illustrated in Table 9. Some synthetic tools have unclear descriptions, some should be divided into clearer sub-functions, and some have overlapping or ambiguous functional boundaries. Proxy tools, however, correspond to actual code implementations, so such problems do not occur.

A.4 Mining Proxy Tools Prompt

This section presents the prompt template for mining proxy tools.

Table 9: Below are several tools related to symptom in APIbench along with their corresponding descriptions.

API Name	**Description**
symptom_analysis	Retrieve an analysis of a patient's medical symptoms over a period of time. Takes in two parameters: 'patient_id' to identify the patient and 'time_period' to specify the time period over which the analysis should be conducted.
symptom_checker	Checks symptoms reported by the patient and provides guidance on whether to seek medical attention. Contains the symptom name and severity.
Symptom_Checker	Get a list of possible mental health conditions based on a user's reported symptoms.
symptom_entry	Log a new record of a medical symptom with user's identifier, symptom name, severity level, and date and time of occurrence.
symptom_query	Search for all symptom records associated with a user based on start and end date.
SymptomAnalysis	This API is used to analyze the recorded medical symptoms of a user over a period of time. 'start_date' and 'end_date' parameters are required as input, which represent the start and end dates for analyzing the medical symptoms.
symptomChecker	Get medical suggestions based on user symptoms.
SymptomList	Retrieve a list of symptoms that can be tracked.

Prompt for Mining Proxy Tools

Please act as a code evaluation expert and assess whether the provided code is suitable to be used as a tool callable by an Agent based on the following six characteristics. Evaluate the tool's purpose and target audience. The assessment must objectively reflect the code's actual performance. The output format must be JSON.

I. Tool Evaluation Characteristics and Scoring Criteria

1. Clear Invocation Interface

• Definition: Does the code focus on a single, practical function and provide a clear, directly callable interface (e.g., function, class method, API endpoint) with unambiguous parameters and invocation method?

• Definition: Does the code focus on a single, practical function and provide a clear, directly callable interface (e.g., function, class method, API endpoint) with unambiguous parameters and invocation method?

• Output: 1 (Yes) / 0 (No). The explanation must clearly state the core function and interface form (e.g., "Function: Weather query, Interface: get_weather(city: str)").

2. Ability to Acquire External Information

• Definition: Is the tool's core function to acquire external information or data (e.g., calling an API, querying a database, web scraping) to serve the Agent's information needs?

• Definition: Is the tool's core function to acquire external information or data (e.g., calling an API, querying a database, web scraping) to serve the Agent's information needs?

• Output: 1 (Yes) / 0 (No). The explanation must specify the primary data or information source (e.g., "Acquires data by calling a public weather API").

3. Ability to Alter the Environment

• Definition: Is the tool's core function to perform operations that change the state of the external environment (e.g., modifying files, sending emails, controlling hardware, operating a database)?

• Definition: Is the tool's core function to perform operations that change the state of the external environment (e.g., modifying files, sending emails, controlling hardware, operating a database)?

• Output: 1 (Yes) / 0 (No). The explanation must briefly describe the primary operation (e.g., "Submits data to a specified API endpoint to update system status").

4. No Human Intervention Required During Execution

• Definition: Does the tool require human intervention during the execution of its core process (e.g., manual input, confirmation, intervening in logic branches)? An ideal Agent tool should be able to complete the entire process automatically based on input parameters.

• Definition: Does the tool require human intervention during the execution of its core process (e.g., manual input, confirmation, intervening in logic branches)? An ideal Agent tool should be able to complete the entire process automatically based on input parameters.

• Output: 1 (Yes, can complete automatically) / 0 (No, requires human intervention). The explanation must indicate any steps requiring human participation (e.g., "requires confirmation in the console", "process halts due to missing parameters without default values").

5. Clear Feedback

• Definition: After execution, does the tool provide the caller with clear, structured, and parseable feedback (e.g., a result object, status code, clear success/error messages)? Merely printing logs without programmatic return is considered unclear.

• Definition: After execution, does the tool provide the caller with clear, structured, and parseable feedback (e.g., a result object, status code, clear success/error messages)? Merely printing logs without programmatic return is considered unclear.

• Output: 1 (Yes, feedback is clear) / 0 (No, feedback is missing or unusable). The explanation must describe the feedback form (e.g., "returns a dictionary containing 'status': 'success', 'data': ..." or "only prints logs, no return value").

6. Usage Scenario Beyond Low-Level Implementation Details
• Definition: Does the tool encapsulate business functionality with a certain level of logic, making it usable for building higher-level tasks (e.g., "schedule planning", "data analysis"), rather than being merely a low-level technical operation (e.g., "memory address translation", "byte stream processing")?
• Definition: Does the tool encapsulate business functionality with a certain level of logic, making it usable for building higher-level tasks (e.g., "schedule planning", "data analysis"), rather than being merely a low-level technical operation (e.g., "memory address translation", "byte stream processing")?
• Output: 1 (Yes, applicable to high-level scenarios) / 0 (No, belongs to low-level detail implementation). The explanation must briefly describe its applicable high-level scenarios or the reason for judging it as low-level.
II. Audience and Applicability Analysis
• Primary Tool Purpose: Summarize the tool's core purpose in one or two sentences.
• Primary Tool Purpose: Summarize the tool's core purpose in one or two sentences.
• Applicable Scenario Analysis: Based on the above scores, briefly analyze the types of Agent tasks or systems (e.g., information assistant, automated workflow, monitoring system) where this tool is most suitable, or explain why it is not suitable.
• Applicable Scenario Analysis: Based on the above scores, briefly analyze the types of Agent tasks or systems (e.g., information assistant, automated workflow, monitoring system) where this tool is most suitable, or explain why it is not suitable.
III. Output Format

```
{
    "Tool Evaluation Result": {
        "Clear Invocation Interface": {
            "Score": 0,
            "Explanation": ""
        },
        "Ability to Acquire External Information": {
            "Score": 0,
            "Explanation": ""
        },
        "Ability to Alter the Environment": {
            "Score": 0,
            "Explanation": ""
        },
        "No Human Intervention Required During Execution": {
            "Score": 0,
            "Explanation": ""
        },
        "Clear Feedback": {
            "Score": 0,
            "Explanation": ""
        },
        "Usage Scenario Beyond Low-Level Implementation Details": {
            "Score": 0,
            "Explanation": ""
        },
        "Tool Total Score": 0,
        "Primary Tool Purpose": ""
```

905

```
    },
    "Audience and Applicability Analysis": {
        "Applicable Scenario Analysis": ""
    }
}
```

A.5 Constructing Synthetic Tool Playbook

This section presents the prompt template for generating synthetic tool playbook.

Prompt for Generating Tool Playbook

You are a professional technical documentation expert. Now you need to generate a comprehensive, detailed, and complete API technical specification document for an API.
First, the [tool to be introduced] is provided to you:
The interface specification of [tool to be introduced]:
[START_API_DESCRIPTION]{api_description}[END_API_DESCRIPTION]
The implementation code of [tool to be introduced]:
[START_REFERENCE_CODE]{reference_code}[END_REFERENCE_CODE]
The design document of [tool to be introduced]:
[START_DESIGN_DOC]{design_doc}[END_DESIGN_DOC]
Additionally, [other tools in the tool list] are provided:
[START_OTHER_TOOLS]{other_tools}[END_OTHER_TOOLS]
Now you need to provide a detailed introduction to this [API tool to be introduced] from the following perspectives:
Based on the interface specification, implementation code, and design document of [the API tool to be introduced], and in connection with [other tools in the tool list], generate a comprehensive, detailed, and complete API usage specification document. The document content should include the following, output in markdown format, using {language} language:
(Title derived from API functionality summary)
1. Basic Information Introduction
- a) Please state the name of the tool and its main functions (described in concise and clear language). For example: Tool name is "File Format Converter", main function is to convert PDF format files to common formats like Word, Excel.
- b) List all parameters of the tool, including parameter names, data types, and whether they are required. For example: Parameter "source file path", data type string, required; Parameter "target format", data type string, required; Parameter "save path after conversion", data type string, optional.
- c) Describe the return value of the tool, including the return value format, field types, meanings, etc. For example: Return value is in JSON format, containing converted file path, conversion status, and other detailed information.
2. Background Information
- a) Describe the target audience and relevant background information served by the tool. For example: What types of files are involved (PDF, Word, Excel), their characteristics, suitable office scenarios; what frameworks are generally needed to process them.
- b) Explain the code methods used to implement the tool's functionality. For example: calling libraries, implementation logic, data flow, format conversion, etc.
- c) Explain the position of this tool within the entire tool list and its relationship with the overall list. For example: This tool belongs to the "File Processing Tool Group", working together with "File Compression Tool" and "File Encryption Tool" in the same group to provide users with a complete file processing solution.
3. Calling Instructions
- a) Indicate when users should call this tool, and list 3 suitable user requirement scenarios for calling this tool, as well as 3 scenarios that are not suitable for calling this tool.
- b) Introduce the format requirements for parameter filling, design 3 correct parameter filling examples that meet the requirements, and 3 incorrect parameter filling examples that do not meet the requirements.
4. Tool Relationships
- a) Briefly introduce other tools in the tool list and explain the relationships between the current tool and these other tools (such as collaboration, complementarity, substitution, irrelevance, etc.)

- b) Exhaustively list as many functions as possible that can be achieved by combining this tool with other tools in the list in different orders, and provide detailed explanations of each combination's function and application scenarios.

5. Error Analysis

- a) Analyze scenarios where errors may occur when calling this function, and explain how to correct the calling method after an error is reported. For example: When calling, if "source file does not exist" error occurs, it may be due to incorrect source file path; correction method is to check and fill in the correct absolute path of the source file.
- b) Analyze scenarios where the function returns error messages in the operating environment, and explain what actions should be adjusted when encountering such situations. For example: If the operating environment returns "insufficient memory" error, it may be because the files being converted simultaneously are too large or too numerous; adjustment actions could be closing other memory-consuming programs or converting files in batches.

Please ensure not to save space, do not delete any content, and do not omit any content due to length restrictions. Please generate all content of the technical specification document completely and in detail. Use {language} language for output.

Please think carefully step by step and generate a detailed technical description document.

The following is the generated API technical specification document in {language} language:

B Data Example

Figure 7 shows splicing templates and structure of the playbook as well as a complete example of tool playbook, with specific content presented in Markdown format.

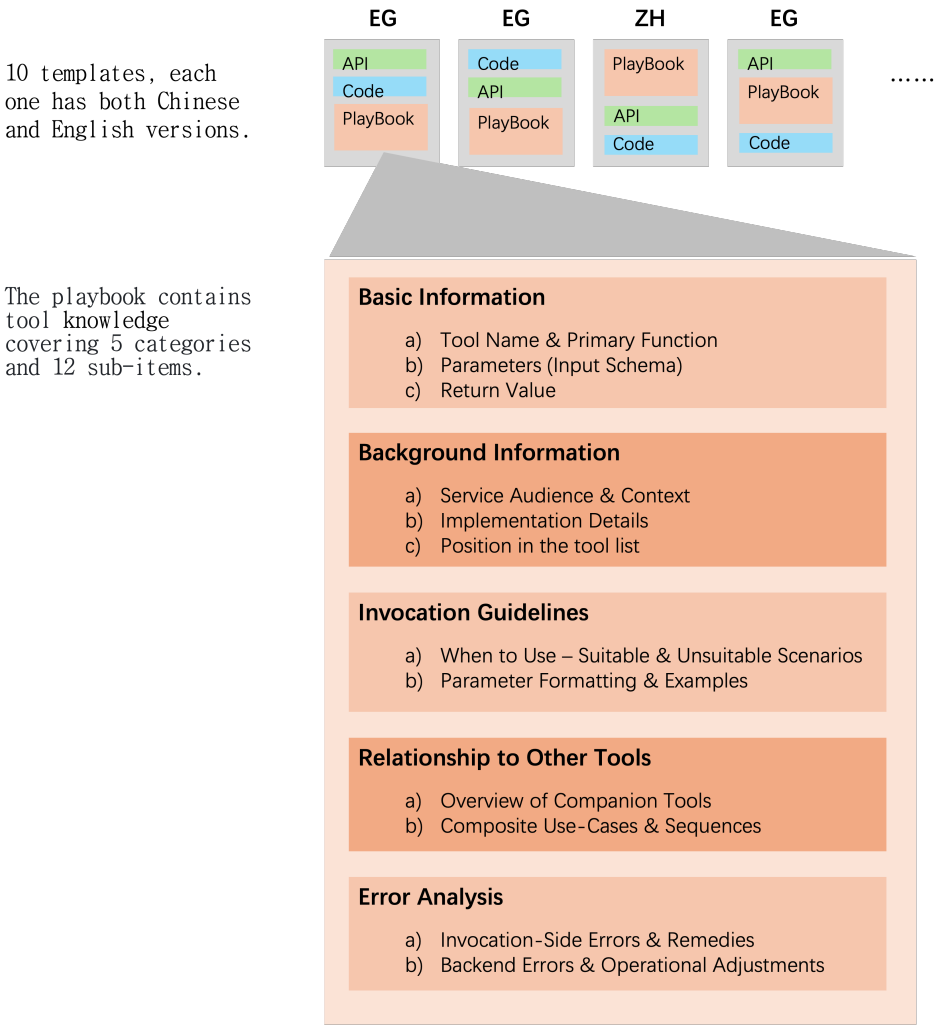

Figure 7: This figure presents ten splicing orders and examples of multilingual splicing templates, illustrating various types of knowledge in the playbook.

Examples of the data

MYPHONEBOOK Tool Complete Handbook: Key Knowledge Points Overview
This article focuses on tool knowledge—the prerequisite for agents to correctly select and call tools. In LLM-based Agent systems, tools such as myphonebook (functional description: Retrieves the full phonebook from the backend service. This tool provides a snapshot of all stored contacts, allowing the caller to view current name-phone mappings) are key to extending agent capabilities. This article directly presents the source code, documentation, and MCP call interface of the myphonebook tool to provide a comprehensive understanding.
The source code reveals the tool's implementation logic and parameter structure, forming the foundation for understanding.
The following is the source code for the myphonebook tool:
The code is implemented in python language.

```
def myphonebook():
    request = requests.get("http://127.0.0.1:5000/myphonebook")
    return request.json()
```

The tool documentation details the functionality, usage, and constraints of the myphonebook tool, complementing the source code.
This article presents the documentation for the myphonebook tool to allow comparison and verification with the code.
Phonebook Snapshot Retrieval – myphonebook Tool
ONE. Basic Information
a) Tool Name & Primary Function
• Name: myphonebook
• Primary Function: Retrieves the complete phonebook from the backend service in a single request, returning a snapshot of all stored contacts (name-phone mappings). It is a read-only operation that provides the caller with a current view of the entire address book.
b) Parameters (Input Schema)
Parameter Data Type Required? Description
None – No The tool does not accept any input parameters. It simply issues a GET request to the predefined endpoint.
Note: The input schema in the tool definition is an empty JSON object, indicating that the tool is invoked without any arguments.
c) Return Value
• Format: JSON object (Python dict after requests conversion)
• Typical Structure

```
{
  "contacts": {
    "Alice": "+1-555-0100",
    "Bob": "+1-555-0101",
    "...": "..."
  }
}
```

• Fields
• contacts (object): A mapping where each key is a contact name (string) and each value is the associated phone number (string).
• Success: HTTP200 with a JSON body as shown above.
• Failure: Non-200 HTTP status codes will raise an exception in the underlying requests.get call (e.g., requests.exceptions.HTTPError). The tool does not handle errors internally; callers must catch them.

TWO. Background Information
a) Service Audience & Context
• Target Users: Applications or agents that need a full view of the phonebook, such as UI dashboards, analytics modules, or batch export scripts.
• Domain Context: The phonebook is a simple key-value store (name → phone number) hosted behind a RESTful service at http://127.0.0.1:5000/myphonebook. It is typical for small-scale contact management systems, internal company directories, or prototype chat-bots.
• Related Data: Each contact is a plain string name paired with a string phone number, no additional metadata (e.g., email, address) is included in this snapshot.
b) Implementation Details
• Library Used: Python's third-party requests library for HTTP communication.
• Logic Flow
1. Compose Request – A GET request is sent to the hard-coded endpoint http://127.0.0.1:5000/myphonebook.
2. Send & Receive – requests.get blocks until the response arrives.
3. Parse JSON – The response body is decoded via response.json(), automatically converting JSON into a Python dictionary.
4. Return – The parsed dictionary is returned to the caller.
• Data Flow:
myphonebook() → HTTP GET → Backend Service → JSON (name-phone map) → Python dict → Caller
• No Authentication / Headers: The snippet does not include authentication tokens, custom headers, or query parameters, implying a trusted local environment.
c)Position in the tool list
The interface definition of the Function Call paradigm for other tools in the tool list is as follows:

```
[
  {
    "name": "get_phone",
    "description": "Fetches the phone number for a given contact
    name from the phonebook. Useful for lookup operations.",
    "inputSchema": {
      "type": "object",
      "properties": {
        "name": {
          "type": "string",
          "description": "The name of the contact whose phone
          number is requested. Must match exactly the name
          stored in the phonebook; case-sensitivity follows
          the backend's rules."
        }
      },
      "required": [
        "name"
      ]
    }
  },
  {
    "name": "delete_phone",
    "description": "Removes a contact from the phonebook by name.
    After successful execution the contact will no longer appear
```

```
    in subsequent queries.",
    "inputSchema": {
      "type": "object",
      "properties": {
        "name": {
          "type": "string",
          "description": "The name of the contact to delete.
          Must correspond to an existing entry; if the name
          does not exist the backend may return an error."
        }
      },
      "required": [
        "name"
      ]
    }
  },
  {
    "name": "update_phone",
    "description": "Updates the phone number of an existing
    contact. If the contact does not exist, the backend may
    create it or return an error, depending on its logic.",
    "inputSchema": {
      "type": "object",
      "properties": {
        "name": {
          "type": "string",
          "description": "The name of the contact whose phone
          number will be updated. Must be an existing entry
          for a pure update operation."
        },
        "phone": {
          "type": "string",
          "description": "The new phone number to associate
          with the contact. Must follow the same formatting
          rules as for adding a contact."
        }
      },
      "required": [
        "name",
        "phone"
      ]
    }
  }
]
```

• Tool Group: “Phonebook Management Tools”.
• Role: Provides a read-only global view, complementing the other CRUD-type tools (add_contact, get_phone, delete_phone, update_phone).
• Interaction Pattern:
• Before: Users may call add_contact / update_phone to modify data.
• After: myphonebook can be used to verify the resulting state or to generate reports.

• Overall Solution: Together, the five tools form a complete API client library for a simple phonebook service, covering creation, retrieval (single and bulk), update, and deletion.
THREE. Invocation Guidelines
a) When to Use – Suitable & Unsuitable Scenarios
Situation Suitable? Rationale (1 sentence)
Generating a full contact report ✓myphonebook returns the entire dataset in one call.
Synchronizing a local cache with the server ✓The snapshot can replace or merge with local storage.
Displaying all contacts on a UI page ✓Efficient for initial page load; pagination must be handled client-side if needed.
Fetching the phone number of a single user X get_phone is more efficient (single-record request).
Adding a new contact X Use add_contact which performs a POST/PUT action.
Removing a contact X Use delete_phone which targets a specific name.
b) Parameter Formatting & Examples
Since myphonebook requires no parameters, the focus is on call syntax.
Correct Invocation Examples
1. Direct Call (synchronous)

```
snapshot = myphonebook()
print(snapshot)
```

2. Within a Function (with error handling)

```
def fetch_snapshot():
    try:
        return myphonebook()
    except Exception as e:
        print("Failed:", e)
        return None
```

3. As Part of a Workflow (e.g., after adding a contact)

```
add_contact(name="Carol", phone="+1-555-0123")
phonebook = myphonebook()
assert "Carol" in phonebook["contacts"]
```

Incorrect Invocation Examples
1. Passing Unexpected Arguments

```
myphonebook(name="Alice")   # TypeError: unexpected keyword argument
```

2. Calling on a Non-existent Endpoint

```
# Modified URL inside the function (illegal - function is hard-coded)
requests.get("http://127.0.0.1:5000/wrongpath")   # Leads to 404
```

3. Assuming Asynchronous Return

```
future = myphonebook()  # Returns dict, not a Future; misuse may cause logic errors.
```

FOUR. Relationship to Other Tools
a) Overview of Companion Tools
Tool Primary Action Relationship to myphonebook
add_contact Create a new name-phone entry (POST/PUT) Complementary – after adding, myphonebook can verify the addition.
get_phone Read a single contact (GET) Partial Overlap – myphonebook provides the same data but for all contacts; get_phone is more efficient for single lookups.
delete_phone Delete a contact (DELETE) Complementary – after deletion, myphonebook shows the updated list.

920

update_phone Update an existing contact's phone number (PATCH/PUT) Complementary – after update, myphonebook reflects the new value.
b) Composite Use-Cases & Sequences
Composite Sequence Resulting Functionality Typical Application
1 add_contact → myphonebook Add a new contact, then fetch the full phonebook to confirm insertion. UI workflow: "AddContact" button followed by refresh of contact list.
2 update_phone → myphonebook Change a phone number, then retrieve the whole phonebook to view updated mapping. Bulk edit operation that needs to display the final state.
3 delete_phone → myphonebook Remove a contact, then obtain the snapshot to ensure removal. Administrative cleanup script.
4 add_contact → update_phone → myphonebook Create a contact, immediately change its number, finally fetch the complete list. Data migration where temporary placeholder numbers are replaced.
5 myphonebook → get_phone (per entry) Pull the whole list, then perform individual lookups locally (e.g., caching). Offline search feature: cache the snapshot once, then query in-memory.
6 myphonebook → delete_phone (loop) Retrieve all contacts, then iterate to delete a subset based on criteria. Bulk deletion based on a naming pattern.
7 myphonebook → add_contact (for missing entries) Get the current snapshot, compare with an external source, add only those not present. Synchronization with an external CRM system.
8 myphonebook → update_phone (batch update) Fetch snapshot, compute new numbers (e.g., formatting), then update each contact. Mass phone-number reformatting (adding country codes).
These sequences illustrate how myphonebook serves as a state-validation or bulk-data source that can be combined with the more granular CRUD tools to build robust workflows.
FIVE. Error Analysis
a) Invocation-Side Errors & Remedies
Error Scenario Likely Cause Fix / Mitigation
requests.exceptions.ConnectionError Backend service not reachable (e.g., server down, wrong host/port). Verify that http://127.0.0.1:5000 is running; start the Flask (or equivalent) server.
requests.exceptions.HTTPError (e.g., 404, 500) Endpoint /myphonebook missing or server internal error. Check server logs; ensure the route exists and returns JSON with status200.
json.decoder.JSONDecodeError Server responded with non-JSON payload (HTML error page, plain text). Confirm that the service sets Content-Type: application/json and returns valid JSON.
TypeError: myphonebook() takes no arguments Caller passed unexpected parameters. Call the function with no arguments.
Timeout (if a timeout is later added) Network latency or server hung. Supply a reasonable timeout argument to requests.get or improve server performance.
b) Backend-Generated Errors & Operational Adjustments
Backend Error Description Operational Response
HTTP500 – Internal Server Error Server threw an exception while building the phonebook snapshot (e.g., database connection failure). Restart the backend, inspect stack trace, ensure the underlying datastore is accessible.
HTTP401 / 403 – Unauthorized / Forbidden Authentication or permission required (not evident in current client code). Add required authentication headers or tokens as per API spec.
Large Payload (e.g., >10MB) causing MemoryError Very large phonebook leads to high memory usage on the client side. Implement pagination on the server, or stream results; on client side, process entries incrementally.
Rate Limiting (HTTP429) Too many requests in a short period. Implement exponential back-off or cache the snapshot for a longer interval.
CORS Policy Violation (in browser contexts) Browser blocks cross-origin GET request. Ensure server includes appropriate Access-Control-Allow-Origin header or use a proxy.
When any of these errors occur, the caller should wrap myphonebook() in a try/except block, log

the exception, and optionally retry after a delay or fallback to a cached version of the phonebook. Based on the source code and documentation of the myphonebook tool, this article provides its structured MCP call interface definition.
This interface clarifies the tool name, parameter specifications, and functional summary.
The following are the interface definition details:

```
{
    "name": "myphonebook",
    "description": "Retrieves the full phonebook from the backend service.
    This tool provides a snapshot of all stored contacts,
    allowing the caller to view current name-phone mappings.",
    "inputSchema": {
        "type": "object",
        "properties": {},
        "required": []
    }
}
```

922