FROM DOCUMENTATION TO ACTION: LEARNING TO REGISTER UNSEEN TOOLS FOR LLM AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Tool use extends the capabilities of large language models (LLMs) beyond text generation to interaction with external systems. Existing approaches primarily supply tool documentation through retrieval or learn to generate tool identifiers. Connecting document-based tool selection to argument generation through a compact interface remains a key challenge. To address this challenge, we introduce **Unseen Tool Registration (UTR)**, a framework that compiles tool documentation into a continuous output embedding and a compact memory. The output embedding is scored against the request hidden state within the model's output layer, enabling a unified next-token decision over tools and text. The selected tool's memory then guides argument generation without placing the full documentation in the task context or requiring further training for newly registered tools. Two-stage training first learns to select tools using document-generated embeddings, then teaches the same decoder to generate arguments from memory while retaining selection ability. Both stages optimize components shared across tools, allowing unseen tools to be registered through one document encoding pass without parameter updates. Evaluations on STQ, ToolE, and ToolBench show that UTR matches or exceeds strong baselines in tool selection. On STQ, UTR achieves 93.43% tool exact match and 74.77% call exact match, improving the latter by 6.29 percentage points over the strong baseline, and reaches 84.67% top-1 selection accuracy on ToolE-OOD. UTR also demonstrates strong performance in large-scale tool selection across tool libraries of varying sizes.

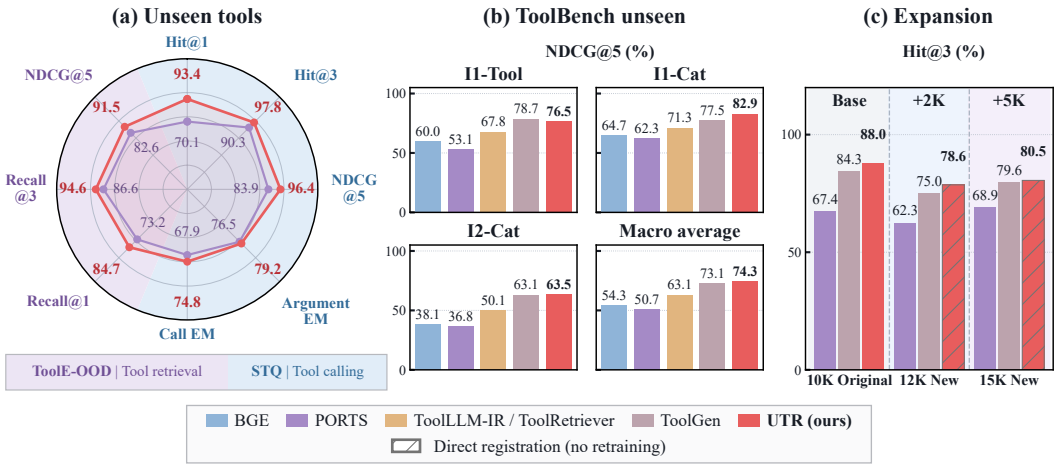

Figure 1: **Left: Unseen-tool generalization.** UTR outperforms strong baselines in tool selection and argument generation on STQ, and tool retrieval on ToolE-OOD. **Middle: Large-scale tool selection.** UTR achieves NDCG@5 comparable to strong trained baselines on ToolBench's official unseen splits (overlapping-API training). **Right: Tool-library expansion.** UTR registers 2K or 5K additional tools on ToolBench without retraining, yet achieves higher Hit@3 than retrained baselines on queries for the newly added tools.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

1 INTRODUCTION

Large language models (LLMs) are increasingly used as agents that interact with the world through external APIs (Yao et al., 2023; Schick et al., 2023; Li et al., 2023). Tool use can turn a language model from a static source of knowledge into a system that retrieves current information and performs concrete actions. Real tool ecosystems, however, are not static: new endpoints appear, existing schemas change, and private APIs may become available only after the model has been trained. A practical agent should therefore be able to acquire an unseen API from its documentation without repeatedly fine-tuning the model or rebuilding its vocabulary. It must identify the correct endpoint, generate valid arguments, and execute the resulting call.

One line of work scales tool use through retrieval. ToolLLM, Gorilla, and Meta-Tool retrieve candidate documents before calling tools (Qin et al., 2024; Patil et al., 2024; Qin et al., 2025). CoTools extends this route to flexible pools of unseen tools using a frozen backbone (Wu et al., 2025). PORTS uses preferences from a frozen LLM to align retrieval with downstream tool use, providing a strong baseline (Molfetta et al., 2025). These systems make document libraries easy to update, but selection remains a separate retrieval step, and candidate documents occupy the request context. Retrieval errors can also propagate to downstream tasks (Shi et al., 2025). Training a dedicated retriever, as in PORTS, introduces an additional optimization cost.

Another line of work internalizes selection as generation. ToolkenGPT learns an output embedding for each tool, while ToolGen assigns each API a unique learned token and generates tool calls within next-token prediction (Hao et al., 2023; Wang et al., 2025). This removes the external retrieval stage, but new tools require new learned representations, and the additional output space grows with the inventory. Toolscaler and ToolWeaver address this scaling problem by encoding tools as compositional sequences of shared discrete codes (Su et al., 2025; Fang et al., 2026). Toolscaler can also encode new tools using its frozen model and existing codebook, without retraining. For methods that learn independent tool identifiers, however, registering new tools requires additional optimization to make those identifiers usable by the generator. Avoiding this repeated training cost motivates representations that transfer directly from documentation to newly available tools.

To address these challenges, we introduce **Unseen Tool Registration (UTR)**. We teach a model to turn documentation into representations for selection and invocation through a process shared across tools. Adding a new tool then requires one document encoding pass and no parameter updates. UTR creates two representations from each document. A dynamic output embedding lets the model choose the tool as part of generating its next token. A compact memory stores the information needed to supply the tool's arguments, such as parameter names and required inputs. After selecting a tool, the same model reads its memory to generate the call. In our calling experiments, just several memory slots per tool support accurate argument generation without inserting the full documentation into the request context. The registry keeps both representations linked to the correct API endpoint.

Our two-stage training teaches these abilities in sequence. The first stage teaches the model to select the right tool using representations built from documentation. The second teaches it to generate arguments from the selected tool's memory while preserving its selection ability. Both stages update lightweight components shared across tools, with the backbone frozen. The model can then apply the same learned process to newly registered tools without further training.

We evaluate whether this shared interface transfers from training tools to new APIs and then to larger tool libraries (Figure 1). On SimpleToolQuestions (STQ), UTR reaches 74.77% complete-call exact match, 6.29 points above the strongest baseline, showing that transfer extends from selection to argument generation. It also achieves 71.51% macro NDCG@5 on ToolBench when training APIs are disjoint from the 10,000 candidates. After registering 5K more APIs without parameter updates, UTR reaches 80.55% Hit@3 on queries for the new tools, exceeding the retrained baselines.

We further test the two parts of the interface separately. Calling ablations show that compact memory retains useful tool information and approaches full-document calling with less context; perturbing memory reveals that its content drives this benefit on ToolBench. A capacity control supports four slots as a practical compact configuration. Finally, an end-to-end development evaluation tests registration at a 47,323-API scale: the full-document calling variant reaches 37.58% SoPR across all 765 tasks. This complements the memory ablations by testing complete task execution with a much larger registry.

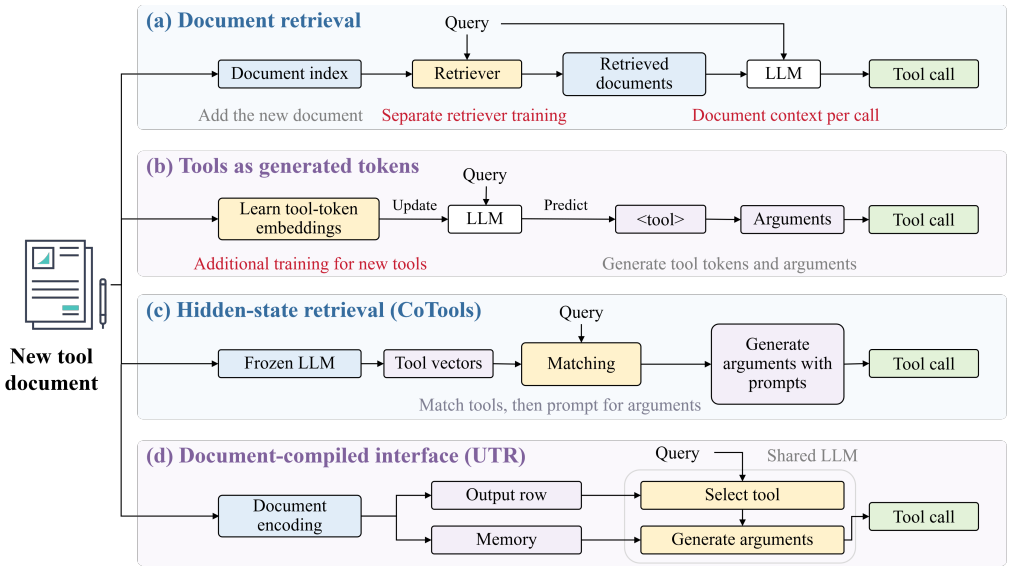

Figure 2: **Interfaces for incorporating tool documentation.** (a) Retrieve documents into the request context. (b) Learn tool-specific tokens for selection and calling. (c) Match hidden-state tool vectors, then prompt for arguments (CoTools). (d) Compile documents into output rows and compact memory for selection and argument generation with a shared decoder (UTR).

Our contributions are threefold. First, we introduce a document-compiled interface that connects direct tool selection to argument generation through an output embedding and compact memory. Second, we develop shared two-stage training that transfers this interface to new tools without tool-specific parameter updates. Third, we demonstrate unseen-tool use and efficient library expansion, identify the role and capacity of memory through controlled experiments, and evaluate complete task execution with over 47K registered APIs.

2 Preliminaries

2.1 Autoregressive generation and hidden states

Hidden states encode contextual information: a request's intended operation, inputs, and history, or a tool document's function and parameter requirements. An autoregressive model uses its final-layer state $\boldsymbol{h} \in \mathbb{R}^d$ to predict the next token. With vocabulary $\mathcal{V}$, output matrix $\boldsymbol{W} \in \mathbb{R}^{|\mathcal{V}| \times d}$, and row $\boldsymbol{w}_v^\top$ for token v, the prediction rule (omitting biases) is

$$p(v \mid \text{context}) = \text{softmax}(\boldsymbol{W}\boldsymbol{h})_v, \qquad z_v = \boldsymbol{w}_v^\top \boldsymbol{h}. \tag{1}$$

The dot product scores an output token against the request state. Whereas ordinary output rows are learned parameters, document hidden states depend on the supplied tool. Mapping these states into output embeddings allows unseen tools to participate in the same selection mechanism.

2.2 Providing tools to a language model

Tools as context. Here, a tool denotes an API endpoint. The model reads its documentation, generates arguments, and receives an execution observation (Yao et al., 2023). Large libraries use retrieval to supply a small set of candidate documents (Qin et al., 2024; 2025), as shown in Figure 2(a). New tools can enter through the document index, but each request requires retrieval and context space. Aligning the retriever with downstream use, as in PORTS, adds separate training (Molfetta et al., 2025).

Tools as generated identifiers. Alternatively, tool selection can be part of next-token prediction. ToolkenGPT learns an output embedding for each tool, and ToolGen assigns each API a unique

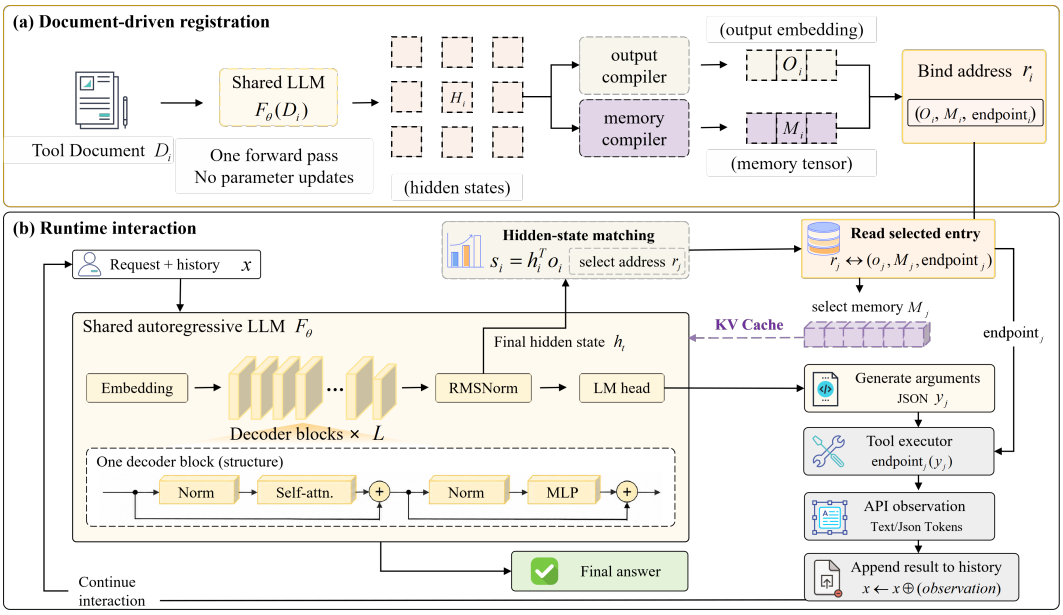

Figure 3: **Overview of UTR. (a) Registration.** Documentation is compiled into an output embedding and a compact memory without parameter updates. **(b) Invocation.** The shared decoder selects a tool, reads its memory to generate arguments, and uses execution feedback to continue or answer.

token and trains the model to generate calls (Hao et al., 2023; Wang et al., 2025). A tool identifier is then scored through its output row in Equation 1, as illustrated in Figure 2(b). This removes the separate retrieval step, but independently learned tool identifiers require additional training when new tools are added.

Hidden-state retrieval. CoTools trains judge and retrieval modules over a frozen LLM's hidden states (Wu et al., 2025). It matches transformed request and document states, then uses in-context prompting to fill the selected tool's parameters (Figure 2(c)). New tools enter through document encoding; selection uses a learned matching interface rather than tool-token output rows. This separates tool matching from prompt-based argument generation.

2.3 Tool use with unseen APIs

Shared document encoders can accommodate tools beyond the training inventory. CoTools does so through hidden-state matching, without learning an independent identifier for each new tool.

Toolscaler converts document representations into sequences of shared discrete codes and trains a generator for tool retrieval and calling (Su et al., 2025). Its extension procedure also encodes new tools with the frozen model and existing codebook, without retraining. UTR shares the goal of transferring document knowledge to new tools, but uses a different interface: a continuous output embedding enables direct selection, and a compact memory supplies information for argument generation (Figure 2(d)). The same decoder uses both representations, connecting document-based matching to invocation without inserting the full tool document into the request context.

3 The Unseen Tool Registration Framework

UTR makes new tools selectable and callable from documentation without retraining the model for each tool. It compiles a document into two complementary representations: an output embedding for selection and a compact memory for argument generation (Figure 3). We first describe how these representations connect registration to tool use, then explain the two-stage training that makes this process transferable.

3.1 FROM DOCUMENTATION TO SELECTABLE AND CALLABLE TOOLS

Document-based registration. Let D_i describe tool i, and let x contain the user request and prior observations. The same language model F_θ encodes them separately, producing document states $\boldsymbol{H}_i = F_\theta(D_i)$ and the request state $\boldsymbol{h} = F_\theta(x)_{|x|}$. Two shared compilers transform these states:

$$\boldsymbol{o}_i = C_{\text{out}}(\boldsymbol{H}_i) \in \mathbb{R}^d, \qquad \boldsymbol{M}_i = C_{\text{mem}}(\boldsymbol{H}_i) \in \mathbb{R}^{S\times d}. \tag{2}$$

The output compiler summarizes the document into a vector for selection. The memory compiler uses S learned queries to retain information from different document positions, including argument names and input requirements. Our calling interface uses several memory slots; Section 4.4 evaluates memory capacity and recommends a four-slot compact configuration. A registry $\mathcal{R}$ binds $(\boldsymbol{o}_i, \boldsymbol{M}_i)$ to a reserved address r_i and the provider's API endpoint. All tools share the same compilers; registering an unseen tool therefore requires one document encoding pass and no parameter updates. Compiler details are given in Appendix A.1.

Selection through next-token prediction. The registered output embedding gives a new tool a score against the current request state, using the mechanism introduced in Section 2:

$$z_{r_i} = \boldsymbol{h}^\top \boldsymbol{o}_i, \qquad p_\theta(r_i \mid x, \mathcal{R}) = \text{softmax}(\boldsymbol{z})_{r_i}. \tag{3}$$

Ordinary text tokens retain their original logits, while unbound tool addresses are masked. Tools and text thus compete in one output distribution. The address identifies an endpoint; its score comes from the document-derived embedding, allowing the model to select a newly registered tool without learning a dedicated output row for it.

Invocation through the selected memory. After selecting r_i, the runtime preserves the request's KV cache and inserts $\boldsymbol{M}_i$ in place of the address embedding. The same decoder then generates JSON arguments y conditioned on the request and selected memory. The resulting call distribution separates selection from argument generation:

$$p_\theta(i, y \mid x, \mathcal{R}) = p_\theta(r_i \mid x, \mathcal{R})\, p_\theta(y \mid x, \boldsymbol{M}_i). \tag{4}$$

The runtime executes the API and appends its observation to the history, allowing the model to continue or generate `Finish`. Only the selected tool's S slots enter the call context. For N tools, registration requires $O(N(S+1)d)$ cached storage and selection scores N rows, while document context per call remains fixed.

3.2 TWO-STAGE TRAINING FOR TRANSFER TO NEW TOOLS

The model must learn both which document-defined action fits a request and how to use that action's memory. We teach selection first, then invocation while retaining selection supervision. Both stages freeze the backbone and optimize lightweight components shared across tools.

Stage 1: align requests with document-defined actions. For candidate addresses $\mathcal{C}$ and relevant addresses $\mathcal{P} \subseteq \mathcal{C}$, the selection objective increases the probability assigned to relevant tools:

$$\mathcal{L}_{\text{sel}}(\mathcal{A}) = -\log \frac{\sum_{r_i \in \mathcal{P}} \exp(\boldsymbol{h}^\top \boldsymbol{o}_i)}{\sum_{v \in \mathcal{A}} \exp(z_v)}, \qquad \mathcal{A} = \mathcal{C} \text{ in Stage 1.} \tag{5}$$

This stage trains the shared output compiler and LoRA adapters (Hu et al., 2021). Varying the candidate sets and address assignments encourages selection from document content rather than memorized tool IDs.

Stage 2: learn to call through memory without losing selection. We add the memory compiler and continue training the shared adapters and output compiler. Teacher-forced argument generation minimizes $\mathcal{L}_{\text{arg}}$, the negative log-likelihood of reference JSON conditioned on the correct memory. $\mathcal{L}_{\text{rank}}$ makes reference arguments more likely under the correct memory than an incorrect tool's memory, while $\mathcal{L}_{\text{schema}}$ supervises parameter names. Together, these objectives encourage the decoder to use the selected tool's information rather than rely on the request alone:

$$\mathcal{L}_{\text{stage2}} = \mathcal{L}_{\text{arg}} + \lambda_r \mathcal{L}_{\text{rank}} + \lambda_s \mathcal{L}_{\text{schema}} + \lambda_t \mathcal{L}_{\text{sel}}. \tag{6}$$

Here $\mathcal{A} = \mathcal{V}_{\text{text}} \cup \mathcal{C}$ includes ordinary text tokens and sampled tool addresses, preserving selection during invocation training. We use $\lambda_r = \lambda_s = 0.1$ and $\lambda_t = 1$; loss definitions and implementation details appear in Appendix A.2. After training, all parameters remain frozen as new tools are registered. Within the reserved address capacity, expansion changes only the registry, without modifying the tokenizer or optimizing new tool-specific parameters.

4 EXPERIMENTS

We first test transfer to unseen APIs and expansion without retraining, then isolate how output alignment and memory support selection and calling. Finally, we evaluate complete task execution with a 47,323-API registry.

4.1 EXPERIMENTAL SETUP

Model and training. UTR uses Qwen3-8B (Yang et al., 2025) with a frozen backbone and shared LoRA adapters and compilers. STQ evaluates both selection and invocation; ToolE and ToolBench evaluate selection. New APIs are registered with all parameters frozen. Appendix A.2 gives the training and generation settings.

Benchmarks and metrics. STQ (Wu et al., 2025) tests selection and argument generation for unseen APIs, using retrieval metrics and normalized exact match (EM) for calls. ToolE (Molfetta et al., 2025) tests retrieval in both in-domain and out-of-domain settings. ToolBench (Qin et al., 2024) tests ranking in a 10K-API library, reporting macro NDCG across the official unseen splits. These benchmarks measure selection and call generation; Section 4.5 evaluates execution and final answers. Dataset sizes and complete results are in Appendices C.1 and B.

Baselines and API exposure. We compare BM25 (Robertson & Zaragoza, 2009), E5 (Wang et al., 2022), BGE (Xiao et al., 2023), Qwen3 hidden states, and tool-oriented methods: ToolLLM-IR/ToolRetriever (Qin et al., 2024), Meta-Tool (Qin et al., 2025), CoTools (Wu et al., 2025), PORTS (Molfetta et al., 2025), and ToolGen (Wang et al., 2025). On ToolBench, we distinguish overlapping-API and disjoint-API training; ToolRetriever, PORTS and UTR receive matched supervision in each condition. ToolGen learns identifiers for the full candidate library and appears in the overlapping block. On STQ, baseline selectors use a Qwen3-8B caller, while UTR invokes tools through its selected memory.

4.2 MAIN RESULTS

Document-based registration transfers to unseen-tool calling. We test whether the learned interface can both select and call APIs absent from training. Table 1 shows that UTR leads all STQ selection and call-generation metrics, reaching 93.43% tool EM and 74.77% complete-call EM. The latter improves on BGE by 6.29 points, so the gain extends to producing the correct arguments, not just retrieving a relevant endpoint. On the separate ToolE-OOD benchmark, UTR reaches 91.47% NDCG@5 versus 82.63% for PORTS, supporting transfer of the same registration principle across benchmarks.

Direct selection scales to a 10K tool library. We next test whether document-derived output rows remain discriminative among many competing APIs. In Table 2, UTR reaches 74.30% macro NDCG@5 under overlapping-API training, comparable to ToolGen's 73.09%. When training and candidate APIs are disjoint, UTR still reaches 71.51%, 18.76 points above the matched ToolRetriever. This result shows that strong large-library selection does not require learning an independent output row for every candidate API. The 2.79-point difference between UTR's overlapping and disjoint settings is also informative: removing direct supervision on candidate APIs leaves most of its ranking performance intact. The shared compiler can therefore transfer the selection interface learned from other tools to the registered library.

4.3 SCALING TOOL REGISTRATION WITHOUT ADDITIONAL TRAINING

Registration makes new tools usable without retraining. To test library growth after training, we independently add 2K or 5K APIs to the same 10K model and evaluate new-tool queries against the entire expanded library. UTR registers their documents with frozen parameters, whereas PORTS and ToolGen receive additional training. Table 3 shows that UTR has the highest Hit@3 in both settings: 78.64% at 12K and 80.55% at 15K. The new APIs therefore become competitive with existing tools through registration alone. ToolGen retains higher NDCG@5 at 15K, so the advantage is in top-three coverage rather than every aspect of ranking. Because each new tool competes against the original library as well as the other additions, this comparison tests whether its compiled output row

Table 1: **Unseen-tool selection and calling.** Scores are percentages; higher is better.

Method	STQ retrieval		STQ call EM			ToolE-ID	ToolE-OOD	
	H@1	N@5	Tool	Arg.	Joint	N@5	R@1	N@5
Qwen3 states	12.20	19.38	25.70	72.70	20.08	7.83	8.41	20.18
BM25	71.58	80.63	82.08	76.08	63.23	34.72	33.72	44.97
E5-large	78.42	88.13	87.05	76.92	67.82	57.92	58.58	73.28
BGE-large	82.18	90.55	88.84	76.83	68.48	68.57	69.53	81.92
ToolLLM-IR	66.04	78.59	83.77	76.45	65.01	57.29	60.41	72.69
Meta-Tool	60.04	75.01	79.74	76.27	62.29	68.54	71.55	82.05
CoTools	10.51	19.36	27.39	70.36	21.11	24.84	9.91	29.26
PORTS	70.08	83.90	87.52	76.55	67.92	84.43	73.17	82.63
UTR	**93.43**	**96.38**	**93.43**	**79.17**	**74.77**	**88.93**	**84.67**	**91.47**

H/N/R: Hit/NDCG/Recall. E5/BGE abbreviate E5-large-v2/BGE-large-en-v1.5. Full metrics and reference calls: Appendix B.

Table 2: **Selection from 10K ToolBench APIs.** NDCG scores (%); higher is better.

Method	I1-Tool	I1-Cat	I2-Cat	OOD macro		
	N@5	N@5	N@5	N@1	N@3	N@5
Training-free baselines						
Qwen3 states	0.00	0.31	0.00	0.17	0.10	0.10
BM25	48.61	46.47	35.61	47.33	40.42	43.56
E5-large	60.88	61.25	36.94	56.83	49.84	53.02
BGE-large	60.03	64.73	38.07	56.00	50.53	54.28
Overlapping APIs (3,721; ToolGen: 10K)						
ToolRetriever	67.79	71.30	50.08	61.67	58.90	63.06
PORTS	53.07	62.31	36.78	52.50	46.49	50.72
ToolGen	**78.69**	77.49	63.09	73.50	70.05	73.09
UTR-overlap	76.53	**82.90**	**63.47**	**76.00**	**70.82**	**74.30**
Disjoint APIs (zero overlap)						
ToolRetriever	60.63	60.46	37.18	54.50	48.97	52.75
PORTS	49.71	56.56	32.88	47.33	42.68	46.39
UTR-disjoint	**73.64**	**79.82**	**61.06**	**76.00**	**67.63**	**71.51**

ToolGen trains on all 10K API identifiers and has no disjoint-API result. Full per-split metrics and gains: Table 7.

fits the existing selection space. Strong coverage after expansion supports this compatibility without updating the previously learned interface.

Avoiding retraining reduces the cost of library growth. The benefit is also practical: at +5K, UTR's registry construction and full evaluation together use 0.042 H100-hours, versus at least 3.11 for the retrained baselines. UTR thus combines competitive new-tool selection with a much cheaper expansion procedure. These are total expansion-and-evaluation costs; Appendix A.3 gives the timing protocol. For a changing tool inventory, the practical distinction is that UTR processes the added documents once and reuses their registered representations on subsequent requests. The expansion experiment thus tests both requirements for useful registration: new tools must be selectable among existing candidates, and incorporating them must avoid another model-training cycle.

4.4 Ablation studies

We test what makes the compiled interface effective: alignment for selection, tool information in memory, and sufficient memory capacity. Appendices A.3 and C give the protocols and additional controls.

Table 3: **New-tool selection and expansion cost.** Queries search the full expanded library.

Method	Updates	H@1↑	H@3↑	N@5↑	H100·h↓
12K (+2K)					
PORTS	Retrained	46.82	62.27	51.59	2.47
ToolGen	Retrained	64.55	75.00	70.52	≈10.77
UTR	None	**65.00**	**78.64**	**70.98**	**0.038**
15K (+5K)					
PORTS	Retrained	51.64	68.91	52.14	3.11
ToolGen	Retrained	**67.09**	79.64	**69.77**	≈15.4
UTR	None	66.18	**80.55**	67.72	**0.042**

Scores are percentages. Costs include successful expansion and full evaluation; full metrics and wall times: Table 8.

Output alignment improves direct tool selection. We test whether document states can serve directly as output rows by replacing the compiler with mean pooling under the same training budget. Figure 4 shows that learning the output mapping improves all three ranking metrics on unseen ToolBench queries. NDCG@1 rises from 73.67% to 76.00%, a 2.33-point gain, indicating better placement of the correct tool at the first decision. A useful document representation thus still benefits from alignment with the decoder's request states. This supports the output compiler as the link between encoding a new tool and making it selectable.

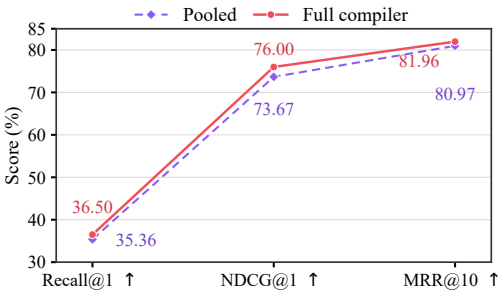

Figure 4: The compiler improves first-choice selection over pooled states.

Memory approaches full-document calling with less context. We compare query-only, memory and full-document callers to test whether the compiled memory preserves information needed for invocation. Giving the caller the correct tool isolates argument generation; sharing tool choices across callers tests complete calls without changes in selection quality. Figure 5 shows that it does: with the correct tool given, memory raises argument EM from 39.15% to 54.47%. With shared tool choices, it reaches 20.00% complete-call EM, close to the full-document caller's 20.85%, while using 26.94% fewer input positions. The benefit is therefore a more compact calling interface that retains much of the utility of tool documents. The full document still helps when the correct tool is given, but most of its complete-call performance is retained by the compact memory under the same selector.

Perturbing memory reveals the value of its tool information. To test whether the caller uses the stored content, we replace the selected tool's memory with another tool's memory while keeping its size fixed. Figure 6(a) shows a sharp drop: gold-tool argument EM falls from 54.47% to 16.60%, and complete-call EM from 20.00% to 7.23%. Zeroing the same slots performs still worse. Thus, the ToolBench caller benefits from tool-specific information in memory, beyond simply receiving additional input positions. Appendix C.3 reports the complementary STQ control, where content sensitivity is weaker.

Four slots provide a practical compact configuration. We vary memory capacity to identify a compact choice for practical use. In a separate comparison using reference histories and plans (Figure 6(b)), four slots reach 40.30% complete-call EM, the best result among the tested memory sizes; increasing to 8–16 slots provides no observed gain. Full documents reach 42.79%, leaving a 2.49-point gap with four slots. These results favor four slots when a compact calling interface is desired: they preserve the strongest memory result while minimizing its input footprint. We recommend this configuration for deployment, while retaining full documents as an option when the additional context is acceptable. Appendix C.5 gives the detailed control.

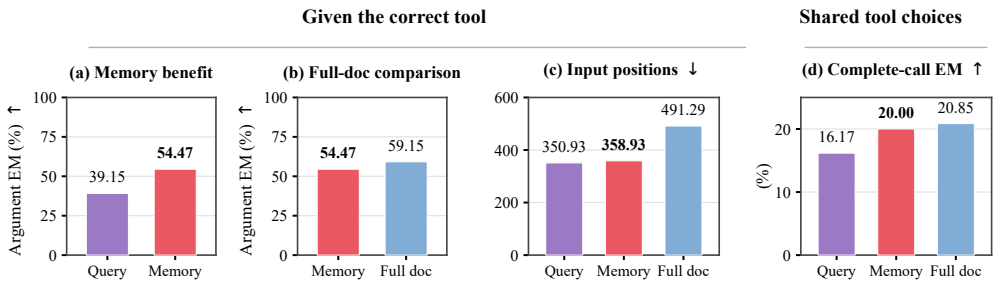

Figure 5: **Compact memory for ToolBench calling.** (a,b) Gold-tool argument EM; (c) input positions; (d) complete-call EM with shared tool choices. No execution.

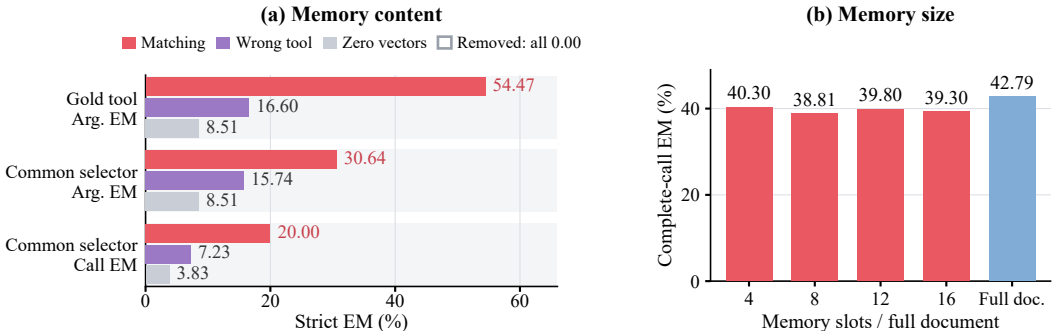

Figure 6: **Memory content and capacity.** (a) Perturbing memory tests reliance on tool information. (b) Four slots offer a compact calling configuration. Protocols: Appendix C.

Table 4: End-to-end SoPR (%) on all 765 tasks with a 47,323-API registry. The DeepSeek judge gives Unsure half credit; groups are weighted by task count.

Method	I1 (474)	I2 (230)	I3 (61)	All (765)
UTR	37.76	39.35	29.51	37.58
ToolRetriever	19.20	14.78	11.48	17.25

4.5 End-to-end development evaluation

Registration supports end-to-end use of a much larger tool library. We evaluate whether the learned interface remains useful through tool selection, execution and final answering when the registered inventory grows far beyond the positive training pool. That pool covers 7,171 APIs; at evaluation, UTR registers 47,323 APIs with frozen parameters. Table 4 reports all 765 tasks: UTR achieves 37.58% SoPR, compared with 17.25% for ToolRetriever, and scores higher in each of the three task groups. This demonstrates task-level utility at a registry scale well beyond the supervised API inventory. The evaluation uses mixed API exposure and full-document calling; it tests the complete system, while the memory contribution is isolated above. Appendix D gives training histories, caller configurations, the DeepSeek judging protocol and execution cases.

5 Conclusion

UTR learns a shared interface that makes new tools selectable and callable from documentation without tool-specific retraining. Unseen-tool and library expansion experiments support transfer of this interface beyond the training inventory. Calling ablations show that compact memory preserves useful tool information with less context, and the capacity control supports a four-slot configuration. The full-document calling variant further demonstrates end-to-end task execution with over 47K registered APIs. Together, these results support growing tool inventories through document compilation rather than repeated model adaptation.

AI USE STATEMENT

Generative AI assistants, including OpenAI Codex, supported implementation, data adaptation, experiment design, analysis, writing, plotting and reference lookup. The end-to-end study uses model-generated plans and a DeepSeek-based simulator and judge; main-benchmark scoring is deterministic.

REFERENCES

Bowen Fang, Wen Ye, Yunyue Su, Jinghao Zhang, Qiang Liu, Yesheng Liu, Xin Sun, Shu Wu, Jiabing Yang, Baole Wei, and Liang Wang. ToolWeaver: Weaving collaborative semantics for scalable tool use in large language models. In *International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=Ge1DKuzWTO.

Zhicheng Guo, Sijie Cheng, Hao Wang, Shihao Liang, Yujia Qin, Peng Li, Zhiyuan Liu, Maosong Sun, and Yang Liu. StableToolBench: Towards stable large-scale benchmarking on tool learning of large language models. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 11143–11156. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.findings-acl.664. URL https://aclanthology.org/2024.findings-acl.664/.

Shibo Hao, Tianyang Liu, Zhen Wang, and Zhiting Hu. ToolkenGPT: Augmenting frozen language models with massive tools via tool embeddings. In *Advances in Neural Information Processing Systems*, volume 36, 2023. doi: 10.52202/075280-1988. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/8fd1a81c882cd45f64958da6284f4a3f-Abstract-Conference.html.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021. URL https://arxiv.org/abs/2106.09685.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. API-Bank: A comprehensive benchmark for tool-augmented LLMs. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 3102–3116. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.emnlp-main.187. URL https://aclanthology.org/2023.emnlp-main.187/.

Lorenzo Molfetta, Giacomo Frisoni, Nicolò Monaldini, and Gianluca Moro. PORTS: Preference-optimized retrievers for tool selection with large language models. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 10007–10030. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.emnlp-main.507. URL https://aclanthology.org/2025.emnlp-main.507/.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive APIs. In *Advances in Neural Information Processing Systems*, volume 37, 2024. doi: 10.52202/079017-4020. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/e4c61f578ff07830f5c37378dd3ecb0d-Abstract-Conference.html.

Shengqian Qin, Yakun Zhu, Linjie Mu, Shaoting Zhang, and Xiaofan Zhang. Meta-Tool: Unleash open-world function calling capabilities of general-purpose large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 30653–30677. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.acl-long.1481. URL https://aclanthology.org/2025.acl-long.1481/.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In *International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=dHng2O0Jjr.

Stephen Robertson and Hugo Zaragoza. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends in Information Retrieval*, 3(4):333–389, 2009. doi: 10.1561/1500000019.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems*, volume 36, 2023. doi: 10.52202/075280-2997. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/d842425e4bf79ba039352da0f658a906-Abstract-Conference.html.

Zhengliang Shi, Yuhan Wang, Lingyong Yan, Pengjie Ren, Shuaiqiang Wang, Dawei Yin, and Zhaochun Ren. Retrieval models aren't tool-savvy: Benchmarking tool retrieval for large language models. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 24497–24524. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.findings-acl.1258. URL https://aclanthology.org/2025.findings-acl.1258/.

Yunyue Su, Jingshuai Zhang, Bowen Fang, Wen Ye, Jinghao Zhang, Bowen Song, Weiqiang Wang, Qiang Liu, and Liang Wang. Toolscaler: Scalable generative tool calling via structure-aware semantic tokenization. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pp. 556–578. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.findings-emnlp.30. URL https://aclanthology.org/2025.findings-emnlp.30/.

Liang Wang, Nan Yang, Xiaolong Huang, Binxing Jiao, Linjun Yang, Daxin Jiang, Rangan Majumder, and Furu Wei. Text embeddings by weakly-supervised contrastive pre-training. *arXiv preprint arXiv:2212.03533*, 2022. URL https://arxiv.org/abs/2212.03533.

Renxi Wang, Xudong Han, Lei Ji, Shu Wang, Timothy Baldwin, and Haonan Li. ToolGen: Unified tool retrieval and calling via generation. In *International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=XLMAMmowdY.

Mengsong Wu, Tong Zhu, Han Han, Xiang Zhang, Wenbiao Shao, and Wenliang Chen. Chain-of-tools: Utilizing massive unseen tools in the CoT reasoning of frozen language models. *arXiv preprint arXiv:2503.16779*, 2025. URL https://arxiv.org/abs/2503.16779.

Shitao Xiao, Zheng Liu, Peitian Zhang, Niklas Muennighoff, Defu Lian, and Jian-Yun Nie. C-Pack: Packaged resources to advance general chinese embedding. *arXiv preprint arXiv:2309.07597*, 2023. URL https://arxiv.org/abs/2309.07597.

An Yang, Anfeng Li, Baosong Yang, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025. URL https://arxiv.org/abs/2505.09388.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

A UTR Implementation Details

A.1 Document compilers

The output compiler pools non-padding document states, $\bar{\boldsymbol{h}}_i = \mathrm{MaskedMean}(\boldsymbol{H}_i)$, then applies a low-rank residual transform and normalization:

$$\boldsymbol{u}_i = \bar{\boldsymbol{h}}_i + \boldsymbol{W}_{\mathrm{up}}\,\mathrm{SiLU}\big(\boldsymbol{W}_{\mathrm{down}}\,\mathrm{LN}(\bar{\boldsymbol{h}}_i)\big), \qquad \boldsymbol{o}_i = \rho\,\boldsymbol{u}_i/\|\boldsymbol{u}_i\|_2. \tag{7}$$

The down/up projections have shapes $k \times d$ and $d \times k$, with $k = 128$; ρ is a learned positive scale. Matching scores the raw request state without cosine normalization. Dynamic logits replace scores at registered addresses while the stored output weights stay fixed; unbound addresses are masked.

The memory compiler uses S learned queries $\boldsymbol{U} \in \mathbb{R}^{S\times k}$ and document keys $\boldsymbol{K}_i = \mathrm{LN}(\boldsymbol{H}_i)\boldsymbol{W}_K$:

$$\boldsymbol{A}_i = \mathrm{softmax}(\boldsymbol{U}\boldsymbol{K}_i^\top/\sqrt{k}), \qquad \boldsymbol{M}_i = T_{\mathrm{mem}}(\boldsymbol{A}_i\boldsymbol{H}_i). \tag{8}$$

Attention excludes padding. T_{mem} applies a rank-128 residual transform, row normalization, and learned slot scales. The memory slots have no fixed field assignments. Both compilers reuse the same document states, requiring only one request-independent document forward pass.

A.2 Training and invocation details

Main-benchmark training and generation settings. UTR uses rank-16 LoRA and rank-128 compilers. Stage 1 trains selection for five epochs at 2×10^{-4}; Stage 2 trains invocation for five epochs at 10^{-4} with several memory slots. On STQ, baseline selectors pass five candidates to a Qwen3-8B caller, while UTR uses its selected memory; generation budgets are 96 and 128 tokens, respectively. Full-identifier and gold-tool calls provide reference conditions.

Stage 1 trains rank-16 attention LoRA and the output compiler. Stage 2 continues both and adds the memory compiler. During invocation training, the target tool token is replaced with its memory and the decoder teacher-forces reference JSON. Prompt tokens are masked; argument loss is the token-weighted batch mean negative log-likelihood.

Let ℓ_i denote the mean token negative log-likelihood of the reference arguments under the correct tool memory, and ℓ_j the corresponding value under an incorrect tool's memory. The ranking loss is $\mathcal{L}_{\mathrm{rank}} = \max(0, m + \ell_i - \ell_j)$, with margin $m = 0.1$. The schema loss supervises parameter names through a JSON object whose values are null. Ranking and selection losses are averaged over examples. Training targets include arguments and `Finish`, rather than free-form thoughts. The registry also retains each endpoint's schema and executor. The cached implementation inserts memory through `inputs_embeds` at the decoder input, extends the attention mask by one entry per slot, and continues sequential cache positions. The vectors pass through all decoder layers. Argument decoding uses the base vocabulary head; dynamic tool rows are inactive, while reserved address IDs remain unmasked. Schema text is stored as registry metadata and is not supplied to the memory-based argument generator. The calling ablations use uncached single-step decoding.

A.3 Controlled ablation protocol

The compiler comparison uses the fixed ToolBench disjoint-API split and 10,000 candidates. Both arms train for five epochs (12,050 updates), with seed 42 and effective batch size 4. The pooled arm bypasses the residual mapping and freezes its LayerNorm and projection weights, while retaining the learned output scale and shared LoRA. We macro-average the three official unseen splits, each containing 200 queries. MRR@10 is computed from the saved top-ten rankings and is not full-ranking MRR.

The ToolBench interface comparison independently trains query-only, memory, and full-document callers from the same Qwen3-8B Stage 1 checkpoint. All use the same 22,859 training examples, five epochs, 3,570 updates, seed 17, effective batch size 32, and learning rate 10^{-4}. LoRA and the output compiler are updated in all arms; only the memory arm trains and uses the memory compiler. Query-only input omits tool information during argument generation, not during prior selection training. The argument, schema, ranking, and selection objectives and weights are retained. Ranking contrasts correct and incorrect documents in the full-document arm; query-only inputs have no corresponding tool-dependent contrast. Full documents are not truncated. The validated padding optimization in the full-document arm preserves the data, training budget, and scoring protocol.

Figure 5 evaluates the same 235 held-out actions, including 164 tool calls and 71 `Finish` actions, under two conditions. Gold-tool evaluation isolates argument generation; common-selector evaluation fixes identical tool choices across the independently trained callers and reports complete-call EM. Both use the same generation and scoring settings, with a 1,024-token generation cap and no additional training. These APIs are seen during training; the queries are held out. Input positions count text tokens and the memory positions where present, excluding offline registration cost. They measure context length, not inference latency. Each arm is a single-seed run.

The memory intervention uses one ToolBench Stage 2 checkpoint trained for five epochs (3,570 updates), with seed 17 and effective batch size 32. The held-out set is split by query and contains 235

actions: 164 tool calls and 71 `Finish` actions. Its APIs are seen during training; this is a controlled invocation diagnostic, separate from the 600-query unseen retrieval benchmark. Gold-tool evaluation fixes the correct action; common-selector evaluation fixes the same Stage 1 selections across all memory conditions (26.81% tool accuracy over 10,000 APIs plus `Finish`). Matching memory belongs to the selected action, even when that selection is incorrect. Wrong memory uses a deterministic permutation excluding the selected action. Zeroing preserves all memory slots; removal drops them entirely. Matching, wrong, and zero conditions retain at least 99.57% JSON validity, whereas removal produces no valid JSON. Its failure therefore includes an input-format shift.

Figures 5 and 6 report strict argument EM: the generated arguments must parse as JSON and equal the reference object without value normalization. Complete-call EM also requires the correct tool. This differs from the normalized EM in the main STQ table. All actions, including `Finish`, remain in the denominator. Results use uncached single-step decoding, with no API execution or judge scoring. Each controlled experiment uses one seed.

For library expansion, the +2K and +5K settings start independently from the same 10K model. UTR takes zero optimizer steps, and checkpoint hashes are unchanged after registration. New-tool queries search the entire expanded library.

Table 8 reports Hit@1/3/5, NDCG@3/5, and Recall@5, which are available under matched definitions for all methods. NDCG@1 duplicates Hit@1 under binary relevance and is omitted. MRR is not compared because the archived runs use different ranking cutoffs. All expansion metrics average queries within the indicated view, rather than macro-averaging the three official unseen splits used in the main ToolBench table.

In the result tables, H/N/R denote Hit/NDCG/Recall, and scores are percentages. Tables 5 and 6 report gains over the strongest baseline in each column. In Table 7, Δ_5 is UTR's macro NDCG@5 minus the row's score within the same training condition; training-free rows are compared with UTR-disjoint. The Updates column in Table 3 refers to additional training during expansion.

Expansion costs include the successful update path and complete evaluation: 1,320 queries against 12K APIs and 1,650 queries against 15K APIs, including the 1,100 original-tool queries in each setting. Selection scores in Table 3 use only the new-tool subsets. All costs use NVIDIA H100 80GB GPUs. PORTS and UTR use one GPU; ToolGen uses four GPUs for training and one for evaluation. H100-hours sum each stage's occupied wall time multiplied by its GPU count. Queueing and unsuccessful attempts are excluded. UTR rebuilds the expanded registry and evaluates it without optimizer updates. ToolGen wall times are approximate to the minute; the displayed H100-hours are rounded.

B COMPLETE BENCHMARK AND EXPANSION RESULTS

Tables 5–8 provide all metrics underlying the compact main tables, including reference calls, gains and wall-clock costs. H/N/R denote Hit/NDCG/Recall. Method abbreviations follow Table 1; higher scores are better and lower costs are better.

Table 5: **Complete STQ results.** Retrieval and normalized call-generation scores (%).

Method	STQ retrieval				STQ call generation		
	H@1	H@3	H@5	N@5	Tool EM	Arg. EM	Joint EM
Qwen3 states	12.20	21.11	26.08	19.38	25.70	72.70	20.08
BM25	71.58	83.77	87.90	80.63	82.08	76.08	63.23
E5-large	78.42	92.68	95.40	88.13	87.05	76.92	67.82
BGE-large	82.18	95.03	96.62	90.55	88.84	76.83	68.48
ToolLLM-IR	66.04	83.77	88.65	78.59	83.77	76.45	65.01
Meta-Tool	60.04	81.89	86.59	75.01	79.74	76.27	62.29
CoTools	10.51	20.83	27.95	19.36	27.39	70.36	21.11
PORTS	70.08	90.34	94.28	83.90	87.52	76.55	67.92
UTR	**93.43**	**97.75**	**98.59**	**96.38**	**93.43**	**79.17**	**74.77**
Δ *vs. best*	+11.25	+2.72	+1.97	+5.83	+4.59	+2.25	+6.29

Δ is the gain over the strongest baseline in each column. Qwen3-8B reference calls (Tool / Arg. / Joint EM, %): all 837 API identifiers, 56.85 / 76.08 / 44.47; correct tool's document, 99.72 / 77.20 / 76.92.

Table 6: **Complete ToolE results.** Retrieval scores (%).

Method	ToolE-ID			ToolE-OOD		
	R@1	R@3	N@5	R@1	R@3	N@5
Qwen3 states	3.71	8.15	7.83	8.41	21.10	20.18
BM25	26.56	36.94	34.72	33.72	47.34	44.97
E5-large	45.11	62.43	57.92	58.58	78.78	73.28
BGE-large	56.12	73.42	68.57	69.53	87.19	81.92
ToolLLM-IR	46.47	61.07	57.29	60.41	77.14	72.69
Meta-Tool	56.08	73.25	68.54	71.55	86.86	82.05
CoTools	11.21	28.35	24.84	9.91	32.30	29.26
PORTS	76.26	88.19	84.43	73.17	86.64	82.63
UTR	**79.14**	**93.69**	**88.93**	**84.67**	**94.56**	**91.47**
Δ *vs. best*	+2.88	+5.50	+4.50	+11.50	+7.37	+8.84

Δ is the gain over the strongest baseline in each column. Method abbreviations follow Table 1.

Table 7: **Complete ToolBench results.** Per-split and macro NDCG scores (%).

Method	I1-Tool N@1	I1-Tool N@3	I1-Tool N@5	I1-Cat N@1	I1-Cat N@3	I1-Cat N@5	I2-Cat N@1	I2-Cat N@3	I2-Cat N@5
Training-free baselines									
Qwen3 states	0.00	0.00	0.00	0.50	0.31	0.31	0.00	0.00	0.00
BM25	49.50	45.43	48.61	49.00	42.00	46.47	43.50	33.81	35.61
E5-large	63.00	58.01	60.88	62.50	57.18	61.25	45.00	34.33	36.94
BGE-large	60.50	57.20	60.03	63.00	59.20	64.73	44.50	35.18	38.07
Overlapping APIs (3,721; ToolGen: 10K)									
ToolRetriever	66.00	64.32	67.79	63.50	65.78	71.30	55.50	46.61	50.08
PORTS	52.50	49.09	53.07	58.00	55.97	62.31	47.00	34.41	36.78
ToolGen	**76.00**	**75.85**	**78.69**	75.50	74.29	77.49	69.00	**60.02**	63.09
UTR-overlap	**76.00**	73.97	76.53	**82.00**	**78.98**	**82.90**	**70.00**	59.52	**63.47**
Disjoint APIs (zero overlap)									
ToolRetriever	61.00	57.72	60.63	62.50	55.93	60.46	40.00	33.24	37.18
PORTS	47.50	45.52	49.71	55.00	51.90	56.56	39.50	30.63	32.88
ToolGen	–	–	–	–	–	–	–	–	–
UTR-disjoint	**73.50**	**70.24**	**73.64**	**80.50**	**75.10**	**79.82**	**74.00**	**57.55**	**61.06**

Method	OOD macro N@1	OOD macro N@3	OOD macro N@5	Gain Δ_5
Training-free baselines				
Qwen3 states	0.17	0.10	0.10	+71.41
BM25	47.33	40.42	43.56	+27.95
E5-large	56.83	49.84	53.02	+18.49
BGE-large	56.00	50.53	54.28	+17.23
Overlapping APIs (3,721; ToolGen: 10K)				
ToolRetriever	61.67	58.90	63.06	+11.24
PORTS	52.50	46.49	50.72	+23.58
ToolGen	73.50	70.05	73.09	+1.21
UTR-overlap	**76.00**	**70.82**	**74.30**	–
Disjoint APIs (zero overlap)				
ToolRetriever	54.50	48.97	52.75	+18.76
PORTS	47.33	42.68	46.39	+25.12
ToolGen	–	–	–	–
UTR-disjoint	**76.00**	**67.63**	**71.51**	–

Δ_5: UTR macro NDCG@5 minus each row's score in the same training condition; training-free rows use UTR-disjoint. ToolGen has no disjoint-API result.

Table 8: **Complete expansion results and costs.** New-tool queries search the full 12K/15K library; scores are percentages.

Method	Updates	H@1	H@3	H@5	N@3	N@5	R@5
12K (+2K)							
PORTS	Retrained	46.82	62.27	70.00	47.85	51.59	58.76
ToolGen	Retrained	64.55	75.00	79.09	**67.84**	70.52	77.27
UTR	None	**65.00**	**78.64**	**83.64**	67.77	**70.98**	**78.22**
15K (+5K)							
PORTS	Retrained	51.64	68.91	74.91	48.61	52.14	57.13
ToolGen	Retrained	**67.09**	79.64	**85.27**	**66.30**	**69.77**	**75.57**
UTR	None	66.18	**80.55**	84.91	64.05	67.72	72.91

Method	Updates	Wall time (h:mm:ss)	H100·h
12K (+2K)			
PORTS	Retrained	2:28:16	2.47
ToolGen	Retrained	≈2:46:00	≈10.77
UTR	None	**0:02:18**	**0.038**
15K (+5K)			
PORTS	Retrained	3:06:49	3.11
ToolGen	Retrained	≈3:55:00	≈15.4
UTR	None	**0:02:30**	**0.042**

Costs include successful expansion and full evaluation on H100 GPUs. ToolGen wall times are approximate; H100-hours account for GPU count. Protocol: Appendix A.3.

C Additional calling and memory controls

C.1 Training and evaluation exposure

Benchmark sizes and scoring. STQ has 10,483 training queries over 999 tools and 1,066 test queries over 837 unseen tools. We report Hit@1/3/5, NDCG@5, and normalized exact match for tools, arguments and complete calls. ToolE follows the supplied split: 4,123 ID queries over 199 candidates and 4,208 OOD queries over 39 candidates, evaluated by Recall@1/3 and NDCG@5. ToolBench uses 10,000 candidate APIs and the official I1-Tool, I1-Cat and I2-Cat unseen splits; NDCG@1/3/5 and macro averages use fixed relevance labels. Overlapping training shares 3,721 APIs with the candidates; disjoint training shares none.

API identities and complete queries define the held-out boundaries used here. Table 9 summarizes the panels. STQ has no normalized cross-split duplicate query or exact duplicate description. Its relation paths share 615 first-level domains and 243 parent paths among the 837 test APIs; these groups are not independent service providers. A lexical screen finds three near-description pairs and 25 test queries with entity-masked templates also present in training. The original benchmark splits are retained.

The available ToolBench registry groups normalized tool/API names. Removing name prefixes leaves 220 repeated document-body groups covering 930 endpoints. The supplied training builder excludes exact matching queries but does not isolate providers or templates. Thus API-ID disjointness and semantic novelty are distinct properties of these splits.

Table 9: Training exposure and evaluation units. Provider, near-endpoint and semantic-template disjointness are not controlled by API-ID splits.

Panel	API exposure	Evaluation unit
STQ main	999 train / 837 held-out APIs	1,066 test queries
ToolBench retrieval	3,721 shared IDs or zero shared IDs	600 queries; 10K candidates
ToolBench calling	Seen APIs	164 ordinary + 71 Finish actions
Slot-count control	Separate development panel	201 ordinary + 128 Finish decisions
End-to-end	598/1,225 relevant IDs are positive training targets; 15 unresolved	765 tasks; 47,323 APIs plus Finish

C.2 Matched-budget memory and document callers

C.2.1 API-held-out full-finetuning control

The first control holds out 200 APIs from the STQ training source, yielding 1,950 development queries. Both callers fully fine-tune the same base model for three epochs, 1,602 updates and 25,593 sample presentations, using seed 17 and the same training order. Each receives the reference API plus four fixed BM25 negatives in the same order. Memory supplies several vectors per candidate; documents supply the five original descriptions. Both use greedy decoding with a 512-token cap and a 1,024-position prompt limit. Training hardware differs between A100 and L20; sample budgets are matched, token budgets are not.

Table 10: API-held-out development calling control. Each column contains the same 1,950 unique queries with the reference API guaranteed among five candidates. Scores are strict exact match (%), without Finish. Intervals are for documents minus memory, using 10,000 paired API-cluster bootstrap samples over 200 APIs (seed 20260923).

Metric	Memory	Documents	Difference, 95% CI (pp)
API selection EM	94.26	97.59	$+3.33\ [1.50, 5.48]$
Argument EM	82.05	82.15	$+0.10\ [-0.62, 0.86]$
Complete call EM	78.05	80.62	$+2.56\ [1.21, 4.07]$
Valid complete format	99.90	100.00	$+0.10\ [0.00, 0.26]$

Documents correct 89 complete calls and regress 39. Mean prompt lengths are 171.35 memory versus 299.24 document positions. Every target is a single-entity array, with 1,420 reference entities appearing literally in the query.

C.2.2 MATCHED-BUDGET LORA CONTROL

The matched control uses the STQ LoRA recipe and the same Qwen3-8B Stage 1 parent, with 1,000 updates, 8,000 presentations, seed 17 and global batch eight. The original 6,550-update cosine schedule and auxiliary loss weights are retained. Both callers receive the same selected tool from a fixed G2 selector, then generate greedily with a 512-token cap; there is no schema repair. Evaluation uses all 1,066 benchmark queries as a development panel, with 837 API identities held out from training. Both arms use the same A100 inference allocation; training hardware and token budgets differ.

Table 11: Matched-budget LoRA calling on 1,066 API-held-out STQ development queries (837 API clusters), with no Finish actions. Strict EM (%) and documents-minus-memory paired 95% API-cluster bootstrap intervals (10,000 samples; seed 20260924). The selector is fixed, not newly trained by this comparison.

Metric	Memory	Documents	Difference, 95% CI (pp)
Fixed API selection EM	94.09	94.09	fixed
Argument EM	75.23	76.27	$+1.03\ [-0.76, 2.82]$
Complete call EM	71.86	73.08	$+1.22\ [-0.47, 2.93]$
Valid output	100.00	99.91	$-0.09\ [-0.29, 0.00]$

Documents correct 46 complete calls and regress 33; both paired intervals include zero. Mean input lengths are 51.86 memory versus 96.27 document positions. These controls use separate checkpoints from the main STQ table. They compare exact calls under single-entity schemas, without API execution.

C.3 TRAINING OBJECTIVES AND MEMORY-CONTENT CONTROLS

A separate STQ LoRA reproduction trains all loss variants for 6,550 updates from the same Stage 1 checkpoint, with one seed. Its Full recipe achieves 94.09% tool EM and 70.73% strict complete-call EM. Removing schema or ranking loss reduces complete-call EM to 69.89% in either case. API-cluster bootstrap 95% intervals for the respective differences are $[-0.65, 2.39]$ and $[-0.75, 2.53]$ percentage points. Removing selection loss while also freezing the output compiler lowers tool EM to 35.08% and complete-call EM to 26.27%; this measures both interventions together. Omitting Stage 1 gives 69.42% complete-call EM, a 1.31-point difference with interval $[-0.82, 3.39]$.

On a separate 200-example STQ development panel, matching and wrong memory both yield 153 correct arguments and 145 complete calls, with identical output sequences on 199 examples. Zeroed memory yields 154 correct arguments and 146 complete calls; removing the slots changes the format and leaves only 19 valid outputs. Memory-content sensitivity is therefore task-dependent, contrasting with the ToolBench intervention in Figure 6(a).

C.4 TEXT/TOOL ROUTING

A frozen compact-memory STQ reproduction is tested on 64 tool queries and 32 synthetic requests answerable without tools. Expanding the registry from 837 to 1,836 APIs lowers reference Top-1 from 62/64 to 56/64 while preserving Top-5 at 64/64. Every direct-answer request activates a tool at both sizes. A follow-up prompt explicitly permitting direct answers still activates tools on all 32 requests. With no registered tools, correct completed answers rise from 14/32 to 17/32 under the revised prompt. The first prompt's maximum tool logit exceeds its maximum text logit by at least 442. The model is trained on positive tool selections; this diagnostic exposes unreliable abstention under the tested prompts. Natural-traffic and Finish calibration are unmeasured.

C.5 MEMORY SLOT COUNT

All five arms train for 500 updates on the same ordered 4,000 effective decisions from 1,877 queries. Four/twelve-slot runs use three L20 GPUs; eight/sixteen/full-document runs use two A100 GPUs. Evaluation uses one seed, 128 development queries and 2,632 candidate tools. Reference causal histories and plans are supplied; plans can reveal APIs, arguments or answers. Tables 12 and 13 separate the 201 ordinary calls from 128 Finish decisions.

Table 12: Slot-count control on 201 ordinary-tool decisions. Values are percentages. Oracle argument EM supplies the reference API; complete-call EM requires both the API and arguments to match.

Interface	Top-1	Top-5	Oracle arg. EM	Complete-call EM
4 slots	47.26	76.12	84.08	40.30
8 slots	45.27	74.13	84.08	38.81
12 slots	46.27	73.63	85.07	39.80
16 slots	46.27	73.63	84.08	39.30
Full document	47.26	76.12	88.06	42.79

Table 13: Finish results on 128 separate component decisions. Entries are counts out of 128. Oracle schema validity supplies the Finish identity and is not answer correctness.

Interface	Top-1	Oracle schema valid	Selection + full EM
4 slots	24	83	2
8 slots	22	4	1
12 slots	25	21	2
16 slots	23	4	2
Full document	25	111	6

Four slots achieve the highest complete-call point estimate among the memory arms and are selected for the compact configuration. Full documents improve by 2.49 points over four slots, with a query-cluster bootstrap 95% interval of $[-1.52, 6.91]$ from 2,000 resamples, seed 17. They correct eleven decisions and regress six. The main calling experiments retain their original compact-memory configuration; this component result is separate from end-to-end evaluation.

C.6 SCOPE OF BASELINE AND EXPANSION COMPARISONS

Table 3 reports new-tool queries; the timed workload also contains 1,100 original-tool queries. Original-tool retention and paired expansion intervals require the corresponding per-query rankings. Reported costs combine registration, any parameter training and evaluation. A frozen-encoder index-only control and separate component timings were not measured.

The CoTools row uses the supplied reproduction checkpoint; its original training command and document-format provenance are unavailable. The available Toolscaler adaptation uses a separate 1K inventory rather than the matched 10K/12K/15K libraries. It is therefore not included in the matched expansion comparison, which reports the PORTS and ToolGen retraining paths alongside UTR registration.

D END-TO-END DEVELOPMENT EVALUATION

D.1 STABLETOOLBENCH-BASED EVALUATION PROTOCOL

UTR uses its saved 5,400-update checkpoint, a 16-slot compiler and a full-document FP32 reader. The reader selects among the top five candidates and receives the selected tool's complete document for argument generation. The runtime uses 32,768 context positions, limits of 768 task-state, 96 plan, 256 argument and 1,024 answer tokens, eight ordinary calls and two validation retries. The StableToolBench-based pipeline uses a DeepSeek simulator and substitute judge (Guo et al., 2024).

All 765 tasks have judgments. Of 1,225 relevant APIs, 598 occur as positive selection targets in the training lineage, with 15 unresolved identities. No normalized complete evaluation query matches a consumed training query. Candidate documents, negative targets and history were not audited to the same depth. This is a development evaluation with mixed API exposure.

The checkpoint contains 43,200 training presentations over 42,832 distinct decision IDs and 31,289 query IDs. The original 21,719-update schedule was not completed and recipes changed along the lineage; convergence was not measured. ToolRetriever trains its retriever using the official implementation on the same 26,360 positive training records (19,513 queries, 24,775 distinct query–API pairs and 7,171 APIs). Five epochs give 131,800 record presentations: 4,120 updates at batch 32. It retains mean pooling and MultipleNegativesRankingLoss. After training, the retriever is frozen and indexes the same 47,323 APIs.

ToolRetriever retrieves five APIs once from the initial query and uses a frozen public Qwen3-8B caller with full schemas and causal interaction history. It generates a thought, a tool choice and arguments in separate stages (96/32/256 tokens; final answers: 1,024 tokens), with two schema retries. It uses greedy non-thinking generation, a 32,768-token context and eight ordinary calls. UTR uses its trained native caller. Both systems use the same executor, simulator and DeepSeek judgment procedure. Table 14 reports the overall comparison; Table 15 breaks it down by task group. Table 4 pools the three G1 groups as I1 (474 tasks), the two G2 groups as I2 (230 tasks), and G3 as I3 (61 tasks).

Table 14: 765-task end-to-end comparison. Credit uses Solved $= 1$, Unsure $= 0.5$, Unsolved $= 0$. Each system has 765 judgments; training and calling configurations are given in Appendix D.

Model	Updates	Solved	Unsure	Credit (%)
UTR best, full documents	5,400	285	5	37.58
ToolRetriever + Qwen3-8B	4,120	132	0	17.25

Table 15: Six-group end-to-end credit (%). TR denotes ToolRetriever, which uses public Qwen3-8B with the calling configuration specified above.

Group	Tasks	UTR credit	UTR (%)	TR (%)
G1 category	153	59.5	38.89	22.88
G1 instruction	163	65.5	40.18	16.56
G1 tool	158	54.0	34.18	18.35
G2 category	124	42.5	34.27	12.90
G2 instruction	106	48.0	45.28	16.98
G3 instruction	61	18.0	29.51	11.48

UTR receives 285 Solved, five Unsure and 475 Unsolved judgments. Half credit for Unsure yields 37.58%; assigning zero or full credit gives 37.25% and 37.91%. The run ends with 654 answers, 75 give-ups and 36 generation errors. Of 2,099 ordinary calls, 1,945 execute successfully and 154 fail. Allocated evaluation cost is 40.09 A100 GPU-hours, including loading and external-service waits. Task success is judged separately from successful API execution.

ToolRetriever receives 132 Solved and 633 Unsolved judgments, with no missing judgments. It ends with 295 answers, four give-ups, 361 generation errors and 105 exhausted call budgets, making 2,708 ordinary tool calls.

D.2 Case studies

The following five UTR trajectories are from G1-category. Each shows the original request, all ordinary tool calls in order, the simulator observations and the final answer. The movie case uses explicitly marked excerpts of long responses and its final answer. Judge labels are the original DeepSeek judgments.

D.2.1 Case 1: Tracking a supplied package identifier

Question (ID 28). I am currently tracking a package with the ID CA107308006SI. Can you provide me with the latest information and localization details of the package? Additionally, I would like to know the country and the type of event associated with the package.
Judge. Solved

Action 1. `packages_v2_track_for_trackingmore_v2`
`{"trackingnumber": "CA107308006SI"}`
Observation 1.
`{'status': 'success', 'data': {'countryCode': 'US', 'trackingNumber': 'CA107308006SI', 'items': [{'event': 'Parcel Collected', 'location': 'LOS ANGELES, CA', 'time': '2023-09-15T10:15:00Z'}, {'event': 'Out for Delivery', 'location': 'LOS ANGELES, CA', 'time': '2023-09-15T12:30:00Z'}, {'event': 'Delivered', 'location': 'LOS ANGELES, CA', 'time': '2023-09-15T14:45:00Z'}]}}`
Final answer (Finish). The latest information for the package with ID CA107308006SI is as follows:
- Country: United States (US)
- Events:
1. Parcel Collected on 2023-09-15 at 10:15:00Z in LOS ANGELES, CA
2. Out for Delivery on 2023-09-15 at 12:30:00Z in LOS ANGELES, CA
3. Delivered on 2023-09-15 at 14:45:00Z in LOS ANGELES, CA

Observation from the trace. The final answer reproduces the country, location and three events in the returned tracking record.

D.2.2 Case 2: Copying invalid UUIDs

Question (ID 12875). I'm creating a database and I need unique identifiers for each entry. Can you help me generate UUIDs?
Judge. Unsolved

Action 1. `generate_uuid_for_helper_function`
`{}`
Observation 1.
`{'result': ['a1b2c3d4-e5f6-7g8h-9i0j-k1l2m3n4o5p6', 'q7r8s9t0-u1v2-w3x4-y5z6-a7b8c9d0e1f2', 'g3h4i5j6-k7l8-m9n0-o1p2-q3r4s5t6u7v8']}`
Final answer (Finish). Here are some UUIDs for your database entries:
a1b2c3d4-e5f6-7g8h-9i0j-k1l2m3n4o5p6, q7r8s9t0-u1v2-w3x4-y5z6-a7b8c9d0e1f2, g3h4i5j6-k7l8-m9n0-o1p2-q3r4s5t6u7v8

Observation from the trace. The returned strings contain non-hexadecimal characters. UTR copies all three strings into its final answer.

D.2.3 CASE 3: COUNTING STEPS WITH UNSPECIFIED PACKAGE IDENTIFIERS

Question (ID 29). My company needs to monitor the progress of multiple packages. Can you help us count the number of steps in the history for each package? This will allow us to optimize our resources and network consumption.
Judge. Solved

Action 1. `count_for_suivi_colis`
`{"colisid": "package1"}`
Observation 1.
`{'colisId': 'package1', 'count': 5}`
Action 2. `count_for_suivi_colis`
`{"colisid": "package2"}`
Observation 2.
`{'colisId': 'package2', 'count': 5}`
Action 3. `count_for_suivi_colis`
`{"colisid": "package3"}`
Observation 3.
`{'colisId': 'package3', 'count': 4}`
Final answer (Finish). The number of steps in the history for each package is as follows:
- Package1: 5 steps
- Package2: 5 steps
- Package3: 4 steps

Observation from the trace. The request supplies no package identifiers. UTR introduces package1, package2 and package3, then reports the returned counts; the judge nevertheless labels the answer Solved.

D.2.4 CASE 4: MOVIE RETRIEVAL WITHOUT STREAMING PLATFORMS

Question (ID 1198). I'm a movie critic and I need to write reviews for the latest movies. Can you provide me with a list of new arrivals on different platforms? It would be great if you could include the streaming platforms and the genres for each movie.
Judge. Unsolved

Action 1. `all_movies_for_movieera_101`
`{}`
Observation 1. (excerpts)
`"status": "ok", "status_message": "Query was successful" [...] "title": "Cheaper by the Dozen" [...] "year": 2003 [...] "genres": ["Action", "Comedy"]`

Action 2. `new_arrivals_for_ott_details`
`{"region": "US"}`
Observation 2.
`{'page': 1, 'results': []}`

Action 3. `new_arrivals_for_ott_details`
`{"region": "IN"}`
Observation 3. (excerpts)
`'genre': ['Action & Adventure'] [...] 'title': 'Jawan' [...] 'genre': ['Comedy'] [...] 'title': 'Patthai' [...] 'genre': ['Thriller'] [...] 'title': 'KGF Chapter 2' [...] 'genre': ['Drama'] [...] 'title': 'Brahmastra Part One: Shiva' [...] 'genre': ['Action & Adventure', 'Animation', 'Fantasy'] [...] 'title': 'Kantara'`

Final answer (Finish). (excerpts) Here are some new arrivals on different platforms:
[...]
1. Title: Jawan
Genre: Action & Adventure
[...]
2. Title: Patthai
Genre: Comedy
[...]
3. Title: KGF Chapter 2
Genre: Thriller
[...]
4. Title: Brahmastra Part One: Shiva
Genre: Drama
[...]
5. Title: Kantara
Genre: Action & Adventure, Animation, Fantasy
[...]

Observation from the trace. UTR makes three calls and reports the titles and genres from the final observation. Its answer does not name a streaming platform for any movie.

D.2.5 CASE 5: EMAIL VALIDATION WITHOUT MX SERVER DETAILS

Question (ID 4415). I need to validate an email address for my company's new website. Can you check if the email 'info@mycompany.com' is valid and deliverable? It would be great if you could also provide the normalized email, domain, and the MX server details.
Judge. Unsure

Action 1. `validate_email_address_for_mailboxlayer {"email": "info@mycompany.com"}`
Observation 1.
{'status': 'success', 'email': 'info@mycompany.com', 'username': 'info', 'domain': 'mycompany.com', 'mx': True, 'syntax': True, 'disposable': False, 'catchall': False, 'smtp': True}

Final answer (Finish). The email 'info@mycompany.com' is valid and deliverable. The normalized email is 'info@mycompany.com', the domain is 'mycompany.com', and the MX server details are available.

Observation from the trace. The response supplies MX and SMTP booleans. The final answer says that MX details are available but gives no MX hostnames or priorities.