SLATE: A Large-Scale Benchmark for Long-Horizon API Selection in E-commerce Tool Use

Rongzhe Wei* Amazon rongzhew@amazon.com

Ge Shi* Amazon gegshi@amazon.com

Min Cheng Amazon mmcheng@amazon.com

Sarthak Ghosh Amazon ghossart@amazon.com

Vaibhav Gorde Amazon gordev@amazon.com

Leman Akoglu Amazon lakoglu@amazon.com

Pan Li Georgia Institute of Technology panli@gatech.edu

Na Zhang Amazon naazhang@amazon.com

Abstract

Enterprises increasingly want LLM agents to automate standard operating procedures over their software stacks—mapping each step of a procedure to the right call among thousands of internal APIs, without hand-authored step-to-API mappings. The core difficulty is *API selection at scale*: choosing and composing the correct tools over a long horizon, judged only by whether the overall task succeeds. Current tool-use benchmarks do not test this regime—they expose only a handful of tools, evaluate single-step invocation, or grade multi-step behavior with subjective LLM-as-a-judge scores that penalize valid alternative trajectories. We introduce SLATE (**S**ynthetic **L**arge-scale **A**PI **T**oolkit for **E**-commerce), a large-scale benchmark that tests large-toolset, long-horizon API selection in a realistic enterprise domain. SLATE couples (i) queries decomposed into hierarchical, tool-grounded procedures over a $\sim$1,000-tool e-commerce library, (ii) a deterministic, context-aware simulator that returns recorded outputs for matching calls and default responses otherwise, and (iii) an end-to-end task-success metric—a task succeeds when it reaches every reference outcome—that credits diverse-but-valid trajectories while remaining fully automatic and reproducible. Experiments with six open decision models and the language models they are built on, from 3B to 12B parameters, show that even the best model completes only 5.0% of the tasks, establishing SLATE as a challenging testbed for reliable enterprise tool use. We release SLATE, its simulator, and the evaluation harness.[2]

1 Introduction

Large language models have become capable tool-using agents that autonomously call external APIs to solve real-world tasks [1, 2], with growing potential to automate complex workflows across e-commerce [3, 4], software development [5], and scientific discovery [6]. Nowhere is this more consequential than inside enterprises. Enterprises increasingly want LLM agents to automate their

*Equal contribution.

[2] https://geshijoker.github.io/slate/

40th Conference on Neural Information Processing Systems (NeurIPS 2026). Workshop: Agentic AI Benchmarks and Applications for Enterprise Tasks.

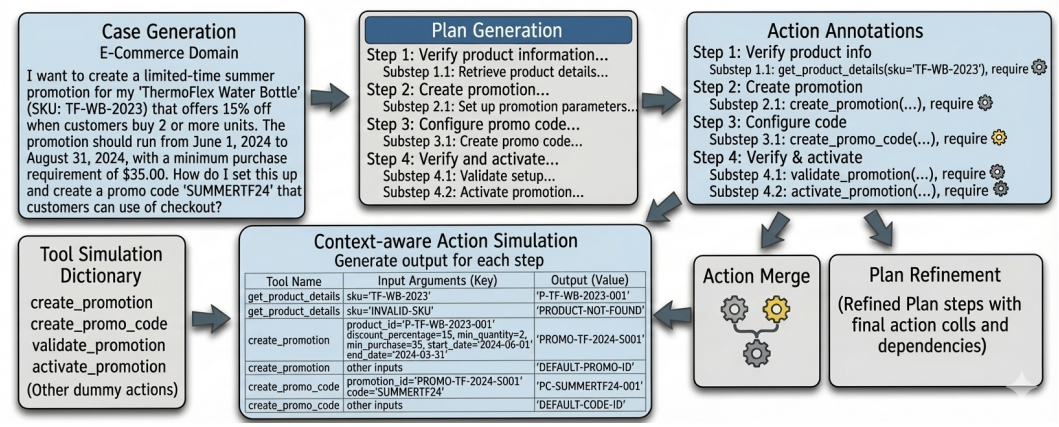


Figure 1: **Illustration of the** SLATE **dataset structure.**

standard operating procedures (SOPs): given a procedure and a large library of internal software-service APIs—often hundreds to thousands—the agent should map each step to the right tool call and execute it, *without* engineers hand-authoring a mapping from every step to a concrete API [7, 8]. This demands agents that select and compose tools reliably over long horizons in large action spaces, precisely the setting of tool-augmented agents in e-commerce operations such as catalog, order, pricing, and customer-service management [3]. Claims of progress on such agents are only as trustworthy as the benchmarks behind them, yet existing tool-use benchmarks do not reflect this regime: small, controlled environments expose only a handful of tools [9, 10]; large tool libraries grade single-step invocation or retrieval rather than long-horizon execution [11–13]; and end-to-end evaluations typically fall back on subjective, non-reproducible LLM-as-a-judge scoring [8, 14]. What is missing is a benchmark that tests large-toolset, long-horizon API selection in a realistic enterprise setting, where success is judged only by whether the overall task completes. We close this gap with SLATE (**S**ynthetic **L**arge-scale **API** **T**oolkit for **E**-commerce), a benchmark designed from the enterprise regime backward. Our contributions are:

- SLATE**, a benchmark for large-scale API selection under end-to-end evaluation.** SLATE is, to our knowledge, the first tool-use benchmark to jointly demand *large tool spaces* (∼1,000 tools), *long-horizon* procedures, and *deterministic tool outcomes*, instantiated in a realistic e-commerce domain (Fig. 1). Its core is a *deterministic, context-aware simulator* that executes every procedure without implementing thousands of live APIs, enabling an *end-to-end task-success metric*: a trajectory succeeds iff it reaches every reference outcome of the task, crediting functionally equivalent solution paths while avoiding the subjectivity and false penalties of LLM-as-a-judge.
- **A benchmarking study.** Evaluating six open decision models and the language models they are built on (3B–12B parameters) shows SLATE is discriminative and that current models fall well short, providing an objective, reproducible yardstick for enterprise API selection.

2 The SLATE Benchmark

SLATE targets the conditions under which enterprise tool-use agents operate: selecting among a large tool library (∼1,000 e-commerce APIs) to execute a long-horizon, hierarchical plan with step-wise deterministic outcomes. Each instance pairs a user query with a hierarchical solution plan whose substeps are grounded to reference tool calls, together with the simulated outputs needed to verify execution (Fig. 1). Instances are produced by a four-stage pipeline—query generation, hierarchical plan generation, tool grounding, and context-aware simulation—followed by manual quality control; we summarize construction below and give full details, including prompt templates, in Appendix A. Existing tool-use benchmarks typically capture only part of the challenge. Small-scale settings such as τ-BENCH [9], ALFWORLD [10], and AppWorld [15] (457 APIs across nine apps) support controlled evaluation but do not reflect large tool spaces, while benchmarks with broader tool libraries often emphasize tool retrieval or single-step correctness rather than verifiable end-to-end execution [8, 16, 13, 11, 12, 14]. Table 1 organizes prior datasets along four complementary properties: *Large Tool Spaces*, *Long-Horizon Execution*, *Deterministic Tool Outcomes*, and *End-to-*

End Outcome Evaluation. SLATE is designed to satisfy all four simultaneously, enabling objective evaluation even when multiple execution trajectories reach the same correct final state.

Table 1: Comprehensive Comparison of Agent Tool-Call Datasets.

Benchmark	Large Tool Spaces (∼1,000 Tools)	Long-Horizon Execution (10+ Steps)	Deterministic Tool Outcomes	End-to-End Outcome Evaluation
SLATE (Ours)	✓	✓	✓	✓
TRAJECT-Bench [17]	✓	✓	✓	×
Toolathlon [18]	×	✓	✓	✓
ToolLLM [8]	✓	×	✓	×
Gorilla [11]	✓	×	✓	×
UltraTool [14]	✓	×	×	×
τ-bench [9]	×	✓	×	✓
τ^2-Bench [19]	×	✓	✓	✓
AppWorld [15]	×	✓	✓	✓
ACEBench [20]	✓	×	✓	×
HammerBench [21]	✓	×	✓	✓
ToolSandbox [22]	×	✓	×	✓
BFCL [23]	×	×	✓	✓
SOP-Bench [24]	×	✓	✓	✓
MetaTool [16]	×	×	×	×
API-Bank [12]	✓	×	✓	✓

2.1 Dataset Construction

SLATE is built by a pipeline of query collection, hierarchical plan generation, tool definition and grounding, tool simulation, and manual post-processing, so that every instance is both logically sound and executable.

Query collection. We target e-commerce because its buyer–seller workflows naturally require coordinating many interdependent APIs. Working with domain experts, we define eleven representative task categories—Pre-purchase Product Questions, Product Search & Comparison, Gifts & Discount Codes, Order & Delivery Changes, and Subscriptions & Returns for buyers, and Product Information Updates, Pricing Changes, Promotions, Fulfillment & Shipping, Inventory & Sales Analytics, and Customer Questions & Reviews for sellers. For each instance we sample a focus area within a category and use Claude to generate and then progressively complicate a query, yielding both linguistic variety and operational complexity. Fig. 2 shows the resulting distribution of library tools across functional categories.

Hierarchical plan generation. Following a top-down, human-like problem-solving paradigm, each query is recursively decomposed into high-level goals, each refined into finer-grained substeps. Crucially, plan generation is performed independently of any toolset, so the decomposition reflects the natural structure of the task rather than the names of available tools, which avoids tool-specific bias in the resulting plans.

Tool definition and grounding. For each substep that requires a tool, we define the tool in a normalized schema specifying its name, input arguments, functionality, and output format, guided by three principles. First, tools must be specific rather than trivially mirror the substep text, so that tool selection remains non-trivial. Second, because execution plans carry strong sequential dependencies, we ensure that later tools take arguments derivable from earlier outputs or from the original query. Third, to promote reuse and avoid redundancy, we embed every tool's name and description with ModernBERT and, for each new query, retrieve the fifty most semantically related existing tools, encouraging the model to reuse them and to create new tools only when genuinely novel functionality is needed. A normalization pass then groups tools into functional categories, merges semantically equivalent ones while reconciling their input and output arguments, and updates the plans accordingly, yielding a coherent, unified plan–tool structure across the dataset.

Tool simulator. Because implementing a thousand live APIs is infeasible, SLATE approximates tool execution with a simulator composed of two modules. A *context-aware simulation database* stores ground-truth traces: during construction, each plan is annotated with its correct tool calls and argument dependencies, and Claude generates a coherent trace of simulated outputs conditioned on the query, the full plan, and prior input–output pairs, stored as a `(tool, arguments)`→`outcome` lookup together with a default response for mismatched calls. A *deterministic matcher* then maps an agent's runtime call to a stored entry beyond exact string comparison, using fixed normalization

Table 2: Key statistics of SLATE.

Property	Value
# tasks (instances)	100
# tools in library	1,000
# relevant tools (used in tasks)	417
# distractor tools	583
Avg. substeps per task	10.8
Substeps per task (range)	7–14
Avg. arguments per tool	3.07
# task categories	11

rules that absorb differences in format (e.g., date conventions) and phrasing; on a successful match it returns the recorded outcome, and otherwise the tool's default response. Every reference solution passes the simulator. This produces realistic, reproducible feedback and lets SLATE measure end-to-end task success automatically. Prompt templates for every stage are provided in Appendix A.

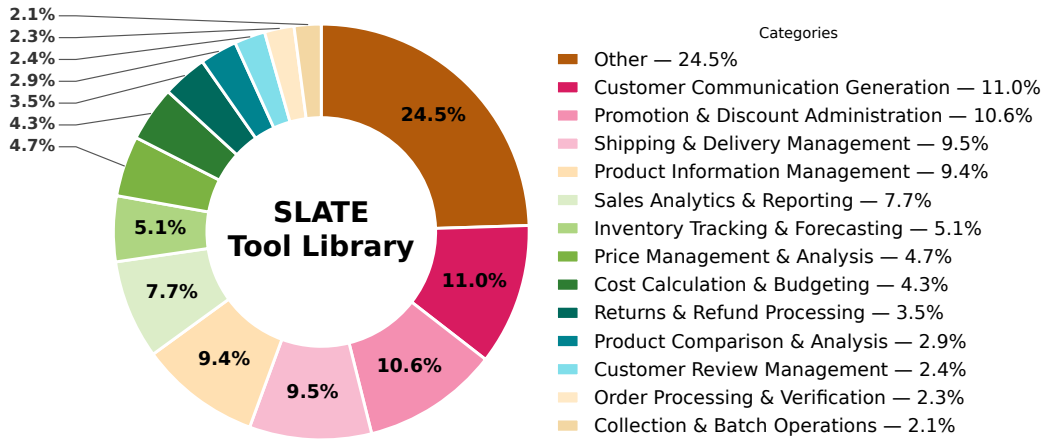


Figure 2: **SLATE tool-library category distribution.**

2.2 Scale and Statistics

Table 2 summarizes SLATE's scale. The benchmark pairs a curated set of complex queries with a tool library of roughly one thousand APIs, of which only a small task-relevant subset is "real" for any given instance while the remainder act as distractors that make selection realistic. Plans are hierarchical and long-horizon, so each instance requires an agent to chain many dependent tool calls whose arguments are derived from earlier outputs. This combination – broad tool space, deep plans, and balanced category coverage across the eleven task categories – is what places instances in the large-toolset, long-horizon regime that motivates the benchmark.

3 Experiments

Setup. The experiments test the task SLATE is built for: mapping each step of a human-centric procedure to the API call that executes it. At every substep the model receives the user query, the earlier substeps with their reference tools and outputs, and the current substep, and selects one of 52 candidates, one per single-character option label (A–Z, a–z), so that the option encoding is identical

Table 3: Task success and step-wise accuracy (%) on SLATE, mean ± std over three option orders. Each decision model is trained from the backbone LLM in the same row.

Backbone LLM	Task success	Step-wise accuracy	Decision model	Task success	Step-wise accuracy
SmolLM3-3B (APO)	0.0±0.0	36.6±0.2	Jevify-SmolLM3-3B	0.3±0.5	52.3±0.3
Qwen3-4B	3.3±0.5	65.7±0.6	AgentDecision-4B	4.3±0.5	72.3±0.2
Gemma-4-E4B-it	1.7±0.5	64.4±0.3	Strands-Decider-E4B	0.0±0.0	44.1±0.5
Tulu-3-8B-DPO	0.3±0.5	42.0±0.6	Jevify-Tulu-3-8B	1.0±0.8	56.1±0.4
Qwen3.5-9B	2.7±0.9	67.7±1.3	OpenJev-Flash-9B	**4.7±1.2**	**74.8±0.8**
Gemma-4-12B-it	**5.0±0.8**	**74.4±0.3**	Strands-Decider-12B	1.0±0.8	56.6±0.9

across models. The candidates always include the reference tool and a "no tool needed" option; the other 50 are the library tools most similar to the substep under Cohere Embed v4 [25], and all 52 are listed in a fixed shuffled order. To isolate API selection, the simulator runs a correctly selected tool with its reference arguments when the steps it depends on were also selected correctly, so one wrong selection breaks every step that depends on it. We report *task success*, the share of tasks that reach every *reference outcome* (the recorded outputs of the final step and of every step whose output no later step uses), and *step-wise accuracy*, the share of substeps with the correct selection, each averaged over three option orders.

Models. A *decision model*, also called a System-1 model after the fast, intuitive mode of dual-process theory [26], is a language model fine-tuned to answer multiple-choice questions directly: given a state, a question, and a list of options, it returns a calibrated probability for every option in a single forward pass instead of generating free-form text [27–30]. We evaluate six open decision models, each paired with the backbone LLM it was trained from; the backbone LLM receives the same candidates and is read from its next-token probabilities over the option labels (Appendix B shows an example). The models span five backbone families from 3B to 12B parameters; checkpoints and prompts are listed in Appendix B.

Results. Table 3 shows that SLATE is far from solved. Even though the reference tool is always among the candidates, step-wise accuracy ranges from 36.6% to 74.8%, and per-step errors compound over the dependent substeps of a task. If each of a task's 10.6 tool steps were selected correctly with independent probability p, the task would succeed with probability $p^{10.6}$, about 4.7% for $p = 0.75$; the best observed task success is 5.0%. Decision models improve step-wise accuracy over their backbone LLMs in four of six pairs (by 6.6–15.7 points), whereas both Strands deciders trail their Gemma backbones (by 17.8 and 20.3 points), so decision-model training does not uniformly help on this task. Results vary little across option orders (standard deviation at most 1.3 points). SLATE is therefore far from saturated for small models, leaving substantial room for future work on reliable long-horizon API selection.

4 Conclusion

We presented SLATE, a large-scale benchmark that tests what enterprise tool-use actually demands: selecting among ∼1,000 APIs over a long horizon, and being graded by a deterministic, end-to-end task-success metric that accepts any valid trajectory rather than matching a single reference trace or relying on a subjective judge. Experiments with open decision models and the language models they are built on (Section 3) show that current models still fall well short, establishing SLATE as a challenging benchmark for reliable enterprise API selection. Beyond ranking agents, SLATE is intended as a testbed for improving them: a natural next step is to study different test-time-scaling strategies that spend rollout compute where it most improves end-to-end success, toward more efficient agents for enterprise API-matching problems. We release the dataset, the context-aware simulator, and the evaluation harness so the community can measure enterprise agents objectively and reproducibly, and extend the construction pipeline to new domains. By isolating API selection at scale under realistic task-level feedback, SLATE provides a reproducible yardstick for enterprise tool-use agents and, we hope, a foundation for developing more reliable and compute-efficient ones. Limitations and ethical considerations are discussed in Appendix C.

References

[1] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

[2] Noah Shinn, Federico Cassano, Beck Labash, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

[3] Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. Webshop: Towards scalable real-world web interaction with grounded language agents. *Advances in Neural Information Processing Systems*, 35:20744–20757, 2022.

[4] Hongyang Yang, Boyu Zhang, Neng Wang, Cheng Guo, Xiaoli Zhang, Likun Lin, Junlin Wang, Tianyu Zhou, Mao Guan, Runjia Zhang, et al. Finrobot: An open-source ai agent platform for financial applications using large language models. *arXiv preprint arXiv:2405.14767*, 2024.

[5] Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 3(4):6, 2023.

[6] Daniil A Boiko, Robert MacKnight, and Gabe Gomes. Emergent autonomous scientific research capabilities of large language models. *arXiv preprint arXiv:2304.05332*, 2023.

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

[8] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789*, 2023.

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

[10] Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. Alfworld: Aligning text and embodied environments for interactive learning. *arXiv preprint arXiv:2010.03768*, 2020.

[11] Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. Gorilla: Large language model connected with massive apis. *Advances in Neural Information Processing Systems*, 37: 126544–126565, 2024.

[12] 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. *arXiv preprint arXiv:2304.08244*, 2023.

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

[14] Shijue Huang, Wanjun Zhong, Jianqiao Lu, Qi Zhu, Jiahui Gao, Weiwen Liu, Yutai Hou, Xingshan Zeng, Yasheng Wang, Lifeng Shang, et al. Planning, creation, usage: Benchmarking llms for comprehensive tool utilization in real-world complex scenarios. *arXiv preprint arXiv:2401.17167*, 2024.

[15] Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. AppWorld: A controllable world of apps and people for benchmarking interactive coding agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL)*, 2024.

[16] Yue Huang, Jiawen Shi, Yuan Li, Chenrui Fan, Siyuan Wu, Qihui Zhang, Yixin Liu, Pan Zhou, Yao Wan, Neil Zhenqiang Gong, et al. Metatool benchmark for large language models: Deciding whether to use tools and which to use. *arXiv preprint arXiv:2310.03128*, 2023.

[17] Pengfei He, Zhenwei Dai, Bing He, Hui Liu, Xianfeng Tang, Hanqing Lu, Juanhui Li, Jiayuan Ding, Subhabrata Mukherjee, Suhang Wang, Yue Xing, Jiliang Tang, and Benoit Dumoulin. Traject-bench:a trajectory-aware benchmark for evaluating agentic tool use, 2025. URL https://arxiv.org/abs/2510.04550.

[18] Junlong Li, Wenshuo Zhao, Jian Zhao, Weihao Zeng, Haoze Wu, Xiaochen Wang, Rui Ge, Yuxuan Cao, Yuzhen Huang, Wei Liu, Junteng Liu, Zhaochen Su, Yiyang Guo, Fan Zhou, Lueyang Zhang, Juan Michelini, Xingyao Wang, Xiang Yue, Shuyan Zhou, Graham Neubig, and Junxian He. The tool decathlon: Benchmarking language agents for diverse, realistic, and long-horizon task execution, 2025. URL https://arxiv.org/abs/2510.25726.

[19] Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. τ^2-bench: Evaluating conversational agents in a dual-control environment, 2025. URL https://arxiv.org/abs/2506.07982.

[20] Chen Chen, Xinlong Hao, Weiwen Liu, Xu Huang, Xingshan Zeng, Shuai Yu, Dexun Li, Shuai Wang, Weinan Gan, Yuefeng Huang, Wulong Liu, Xinzhi Wang, Defu Lian, Baoqun Yin, Yasheng Wang, and Wu Liu. Acebench: Who wins the match point in tool usage?, 2025. URL https://arxiv.org/abs/2501.12851.

[21] Jun Wang, Jiamu Zhou, Muning Wen, Xiaoyun Mo, Haoyu Zhang, Qiqiang Lin, Cheng Jin, Xihuai Wang, Weinan Zhang, Qiuying Peng, and Jun Wang. Hammerbench: Fine-grained function-calling evaluation in real mobile device scenarios, 2025. URL https://arxiv.org/abs/2412.16516.

[22] Jiarui Lu, Thomas Holleis, Yizhe Zhang, Bernhard Aumayer, Feng Nan, Felix Bai, Shuang Ma, Shen Ma, Mengyu Li, Guoli Yin, Zirui Wang, and Ruoming Pang. Toolsandbox: A stateful, conversational, interactive evaluation benchmark for llm tool use capabilities, 2025. URL https://arxiv.org/abs/2408.04682.

[23] Fanjia Yan, Huanzhi Mao, Charlie Cheng-Jie Ji, Tianjun Zhang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. Berkeley function calling leaderboard (bfcl). https://gorilla.cs.berkeley.edu/blogs/13_bfcl_v3_multi_turn.html, 2024.

[24] Subhrangshu Nandi, Arghya Datta, Rohith Nama, Udita Patel, Nikhil Vichare, Indranil Bhattacharya, Prince Grover, Shivam Asija, Giuseppe Carenini, Wei Zhang, Arushi Gupta, Sreyoshi Bhaduri, Jing Xu, Huzefa Raja, Shayan Ray, Aaron Chan, Esther Xu Fei, Gaoyuan Du, Zuhaib Akhtar, Harshita Asnani, Weian Chan, Ming Xiong, Francesco Carbone, and Jeetu Mirchandani. Sop-bench: Complex industrial sops for evaluating llm agents. *arXiv preprint arXiv:2506.08119*, 2025.

[25] Cohere. Cohere Embed v4. https://docs.cohere.com/docs/cohere-embed, 2025.

[26] Daniel Kahneman. *Thinking, Fast and Slow*. Farrar, Straus and Giroux, 2011.

[27] Strands Agents. Strands decider: A small, fast decision model for agentic AI. https://github.com/strands-labs/strands-decider, 2026.

[28] OpenJev. OpenJev-Flash-9B. https://huggingface.co/openjev/OpenJev-Flash-9B, 2026.

[29] JevAgentHarness. JevAgentHarness: Benchmarking, deploying and training decision models that share control with LLMs. https://github.com/JevAgentHarness/JevAgentHarness, 2026.

[30] Praveen Raj U S. Jevify: Turn any open LLM into a calibrated system one decision model. https://github.com/uspraveen/Jevify, 2026.

Appendices

A SLATE Construction

A.1 Prompt Templates

Prompt for Generating Diverse E-Commerce Cases

Generate ONE focused e-commerce query that an {user_type} might have.
User type: {user_type} Focus area: {focus} Complexity level: {complexity}
Requirements based on COMPLEXITY LEVEL:
{complexity} COMPLEXITY GUIDELINES: {self._get_complexity_guidelines(complexity)}
For ALL queries, ensure:

1. Include SPECIFIC DETAILS that can be used as direct arguments in tools:
   - Product names and identifiers (e.g., "Sunset Yoga Mat (SKU: YM-2023-BL)")
   - Order numbers (e.g., "Order #AB-12345678")
   - FULL DATES WITH YEARS (e.g., "January 15, 2023" not just "January 15")
   - Prices with currency (e.g., "$49.99")
   - Specific quantities (e.g., "3 units")
2. Include MULTIPLE DATA POINTS for tools to extract as parameters
3. For COMPLEX and ADVANCED queries, include:
   - Multiple constraints or conditions
   - Timing requirements or deadlines
   - Preferences with priorities
   - Historical context or previous actions
   - Special exceptions or unusual circumstances

Examples of queries at different complexity levels:
SIMPLE: "I need to track my order #RT-78256391 for the Samsung Galaxy Buds that I ordered on May 3, 2023. When will it be delivered?"
MODERATE: "I need to return two items from my Order #112-9384756 placed on March 12, 2023: the Samsung Galaxy S22 with a cracked screen and the protective case. I want to keep the screen protector and charging cable. Can I get a return label for just those two items?"
COMPLEX: "I need to modify my bulk order #BLK-2023-4872 placed on February 28, 2023 for my company. We ordered 50 Lenovo ThinkPad T14 laptops with i7 processors at $1,299 each, but now I need to change 15 of them to the i9 model which costs $1,599 each. We've already paid the deposit of $25,000 and delivery is scheduled for June 15, 2023. I need to know if this change will affect our delivery date and what additional payment is required."

Figure 3: Prompt to generate diverse e-commerce cases from different user perspectives. **Example Variable Values:** `user_type` alternates between "buyer" and "seller" based on query ID; `focus` is randomly selected from buyer focus areas ("product search with multiple conflicting requirements", "order tracking for multiple items with shipping complications", "complex product return with partial refund request", etc.) or seller focus areas ("updating complex product variations and attributes", "inventory management across multiple warehouses", "implementing tiered pricing strategy with conditions", etc.); `complexity` is one of "SIMPLE", "MODERATE", "COMPLEX", or "ADVANCED" with weighted distribution of 0%, 10%, 80%, 10% respectively.

Prompt for Creating Specialized Tool Plans to Resolve E-Commerce Queries (Part 1)

ADVANCED: "I'm managing our company's quarterly office supply order (PO #BZ-45721) placed on April 2, 2023 with scheduled delivery on April 20, 2023 across three locations. For the Chicago office (Location ID: CHI-005), we need to cancel the 12 ergonomic chairs ($259 each) due to their recent merger, but expedite the 15 monitor stands ($89 each) to arrive by April 15. For the Boston office (Location ID: BOS-002), we need to add 8 wireless keyboards ($65 each) and 8 wireless mice ($45 each) for new hires. The New York office (Location ID: NYC-001) shipment is fine as is, but we need to change the delivery window to after 2:00 PM. We're eligible for the 12% corporate discount and already applied the SPRING2023 promo code for 5% off. How will these changes affect our total, and can all these modifications be accommodated before processing begins on April 10?"
Return only the query text with no additional explanation.
Create a plan with specialized tools to resolve this e-commerce query:
User Query ({user_type}, {complexity} complexity): "{query}"
{existing_tools_text}
Please provide:
1. A HIERARCHICAL PLAN with: {plan_requirements}
2. For each SUBSTEP, indicate:
   - If it's a high-level step: "tool": "null" - NO EXCEPTIONS!
   - If no tool is needed: "tool": "No tool required"
   - If a tool is needed: "tool": "tool_name(param1='value1', param2='value2')"
3. CRITICAL: FIRST TOOL CALL arguments must come DIRECTLY from the query
   - Example: If query mentions "Order #AB-12345", first tool should use order_id='AB-12345'
   - Extract actual values from the query text, don't invent new values
4. CRITICAL: SUBSTEP DESCRIPTIONS MUST BE DISTINCT FROM TOOL NAMES
   - Make each substep description MEANINGFUL and CONTEXT-RICH
   - Do not create tool names that are Identical or Very Similar to substep descriptions.
   - BAD: "1.1 Get order details" when using "get_order_details" tool
   - GOOD: "1.1 Retrieve customer's purchase history for Order #AB-123" when using "get_order_details" tool
   - Describe the PURPOSE and CONTEXT of the step, not just the action
   - Include relevant business context and specific goals for each step

Figure 4: Prompt for creating specialized tool plans (Part 1). **Example Variable Values:** `user_type` is either "buyer" or "seller"; `complexity` is one of "SIMPLE", "MODERATE", "COMPLEX", or "ADVANCED"; `query` is the actual query text generated in stage 1. **existing_tools_text:** Dynamically generated list of top 30 most relevant tools from global tool library. **plan_requirements:** SIMPLE (2-3 high-level steps, 1-2 substeps each, 3-5 total); MODERATE (3-4 high-level steps, 1-3 substeps each, 6-8 total); COMPLEX (4-5 high-level steps, 2-3 substeps each, 8-12 total); ADVANCED (5-7 high-level steps, 2-4 substeps each, 12-16 total).

Prompt for Creating Specialized Tool Plans to Resolve E-Commerce Queries (Part 2)

5. TOOL SELECTION AND DESIGN:
   - TRY REUSE EXISTING TOOLS whenever appropriate (from the list above)
   - If none of the existing tools are suitable, design NEW SPECIALIZED TOOLS that:
     - Have SPECIFIC, FOCUSED functionality (not general-purpose)
     - Use SIMPLE arguments (2-3 parameters maximum)
     - Return SIMPLE, FOCUSED results (1-3 fields maximum)
     - Follow snake_case naming convention
     - Avoid creating monolithic "do everything" tools

6. CRITICAL - AVOID TOOL REPETITION:
   - DO NOT use the same tool in consecutive substeps (e.g., avoid "2.1 xxx tool_A; 2.2 xxx tool_A")
   - If you need to call the same tool multiple times, space them out with other operations
   - Each substep should ideally use a DIFFERENT tool to create diversity
   - Create specialized tools for different aspects rather than reusing generic ones
7. IMPORTANT - SEQUENTIAL DEPENDENCIES:
   - Later steps should use results from previous steps for sequential dependencies
   - Use the format OUTPUT_FROM_STEP_X.Y.field to reference previous outputs
   - Example: product_id=OUTPUT_FROM_STEP_1.2.product_id
   - Create a CHAIN of dependencies where each step builds on previous results
   - Ensure that tool outputs from early steps provide necessary inputs for later steps
8. FINAL STEP should produce a DIRECT RESULT that resolves the query
   - The last tool should return a clear outcome or answer
   - For example: confirmation message, success status, or direct result
   - FINAL STEPS should utilize outputs from earlier steps

9. FOR {complexity} COMPLEXITY:
   - {self._get_plan_complexity_guidelines(complexity)}

Figure 5: Prompt for creating specialized tool plans (Part 2). Focus on tool selection, reuse strategy, avoiding repetition, and establishing sequential dependencies between steps.

Prompt for Creating Specialized Tool Plans to Resolve E-Commerce Queries (Part 3)

10. TOOL DIVERSITY REQUIREMENTS:
    - Create tools that span different functional domains (retrieval, validation, processing, notification, etc.)
    - Avoid generic tools like "process_request" or "handle_query"
    - Instead create specific tools like "validate_return_window", "calculate_refund_amount", "send_confirmation_email"
    - Each tool should have a clear, single responsibility

TOOL DESIGN PRINCIPLES:

1. FOCUSED: Each tool should do ONE thing well
2. COMPOSABLE: Tools should work together through their inputs/outputs
3. REUSABLE: Create tools that could be useful in other scenarios
4. SIMPLE: Prefer multiple simple tools over one complex tool
5. DIVERSE: Create tools spanning different functional domains
6. SEQUENTIAL: Design tools to create natural dependencies and data flow

Remember:

- REUSE existing tools whenever appropriate
- FIRST TOOL must use arguments DIRECTLY from the query
- Keep tools SPECIALIZED with SIMPLE inputs and outputs
- Ensure steps are SEQUENTIAL with clear dependencies
- AVOID repeating the same tool in consecutive steps
- FINAL STEP should provide a DIRECT RESOLUTION to the query
- Return valid JSON only

Figure 6: Prompt for creating specialized tool plans (Part 3). Format response as JSON object with "plan" array (high-level steps with "tool": "null", and substeps with tool calls or "No tool required") and "tools" array (defining all tools with "name", "description", "arguments", and "results" fields). **Complexity Guidelines:** SIMPLE (1-2 tool calls, linear, minimal dependencies); MODERATE (2-4 tool calls, at least one dependency and conditional step); COMPLEX (4-7 tool calls, multiple dependencies, validation, branching paths); ADVANCED (7+ tool calls, multi-stage workflow, complex dependencies, edge case handling, maximum tool diversity).

Prompt for Generating Diverse E-Commerce Tools

Generate {batch_size} DIVERSE e-commerce tools with COMPLETELY UNIQUE NAMES.

TOOL NAMING REQUIREMENTS:

1. Each tool name MUST be UNIQUE and use snake_case format
2. NEVER use any of these existing names: {names_str}
3. Create DISTINCTIVE names that avoid generic patterns
4. Consider these name patterns for inspiration: {name_pattern_text}
5. Each tool name should reflect its SPECIFIC FUNCTION and DOMAIN
6. VERIFY that each name is different from all others in your response

TOOL ASSIGNMENTS - Create exactly one tool for EACH of these specific domains: {domains_text}

SAMPLE TOOL STRUCTURE: {json.dumps(sample_tool, indent=2)}

TOOL REQUIREMENTS:

1. Match each tool precisely to its assigned domain above
2. Include 1-3 SIMPLE arguments with clear purposes
3. Return 1-3 FOCUSED result fields
4. Include "query_id": "dummy" in each tool
5. Follow the exact JSON structure of the sample

Return a JSON array containing {batch_size} tools with UNIQUE names:

```
[
  {tool1},
  {tool2},
  ...
]
```

ONLY return the JSON array with no additional text.

Figure 7: Prompt for generating diverse e-commerce tools with unique names. **Example Variable Values:** `batch_size` is configurable (typically 10 or 20, default 10). `names_str` is a comma-separated string of 20 random existing tool names to avoid duplication (e.g., "get_order_details, validate_return_window, calculate_refund, search_products, update_inventory, generate_shipping_label, process_payment, send_notification, track_shipment, analyze_sales, manage_promotions, verify_address, calculate_tax, check_stock, create_invoice, update_customer, generate_report, schedule_delivery, process_return, validate_coupon"). `name_pattern_text` consists of 4 randomly selected naming patterns from: "feature_specific_domain, domain_specific_action, specialized_task_handler, domain_analyzer, action_target_generator, domain_insight_provider, specialized_workflow_automation, target_specific_optimizer". `domains_text` is a numbered list of specific domains (one per tool) from 100 predefined diverse domains covering: Product management (catalog, variants, bundling, photography assessment, competitive analysis, recommendations, sourcing, limited editions, warranties, etc.), Shipping & logistics (delivery, warehousing, returns, tracking, etc.), etc. `sample_tool` is a randomly selected existing tool or default structure containing: `name`, `description`, `query_id`, `arguments` (with type, properties, and argument details), and `results` (with type, properties, and result field details).

Prompt for Categorizing E-Commerce Tools into Functional Domains

Analyze these e-commerce tools and categorize them into functional domains.

TOOLS TO CATEGORIZE: {json.dumps(batch_tools, indent=2)}

PREDEFINED CATEGORIES: {json.dumps(self.categories, indent=2)}

INSTRUCTIONS:

1. For each tool, assign it to the MOST APPROPRIATE category from the predefined list
2. If a tool could fit multiple categories, choose the PRIMARY function
3. Consider the tool's main purpose and typical use case
4. Return a JSON object mapping tool names to their categories

Return format:

```
{
  "tool_name_1": "Category Name",
  "tool_name_2": "Category Name",
  ...
}
```

ONLY return the JSON object with no additional text.

Figure 8: Prompt for categorizing e-commerce tools into functional domains. **Example Variable Values:** `batch_tools` is a batch of 10-20 tools to categorize, each containing `name`, `description`, `arguments`, and `results` fields (e.g., "get_order_details", "calculate_shipping_cost", etc.). `self.categories` is a predefined list of 14 functional categories: "Product Information Management", "Shipping & Delivery Management", "Promotion & Discount Administration", "Sales Analytics & Reporting", "Price Management & Analysis", "Inventory Tracking & Forecasting", "Customer Communication Generation", "Order Processing & Verification", "Product Comparison & Analysis", "Collection & Batch Operations", "Returns & Refund Processing", "Cost Calculation & Budgeting", "Customer Review Management", and "Other".

Prompt for Generating Tool Simulation Outputs

Generate a realistic JSON output for this e-commerce tool call.
QUERY: "{context['query']}"
CURRENT STEP: "{context['step']}"
TOOL:

- Name: {tool_name}
- Description: {tool_def.get('description', 'No description available')}

ARGUMENTS USED: {json.dumps(args, indent=2)}
PREVIOUS TOOL OUTPUTS: {json.dumps(context['previous_outputs'], indent=2) if context['previous_outputs'] else "No previous outputs"}
REQUIRED OUTPUT PROPERTIES: {json.dumps(result_props, indent=2)}
EXAMPLE FORMAT: {json.dumps(example_output, indent=2)}
COMPLEXITY LEVEL: {complexity} {complexity_guidance}
INSTRUCTIONS:

1. Generate SPECIFIC, REALISTIC values consistent with the original query
2. Maintain CONSISTENCY with previous tool outputs
3. Include ALL required properties in the tool's result schema
4. Values should match their expected data types
5. {"This is the FINAL TOOL in the plan. Make sure the output provides a DIRECT RESULT that clearly resolves the user's request (e.g., confirmation, status, or answer)" if is_final_step else "Keep output focused and relevant to the step"}

RETURN ONLY THE JSON OUTPUT OBJECT.

Figure 9: Prompt for generating tool simulation outputs. **Example Variable Values:** `context['query']` is the original e-commerce query text; `context['step']` is the current step description from the plan; `tool_name` is the name of the tool being called; `tool_def` is the tool definition object containing description and other metadata; `args` is a JSON object containing the arguments passed to the tool; `context['previous_outputs']` is a JSON object or array containing outputs from previously executed tools (null or empty if no previous outputs); `result_props` is a JSON schema object defining required output properties with types and descriptions; `example_output` is a sample JSON object showing the expected output format; `complexity` is one of "SIMPLE", "MODERATE", "COMPLEX", or "ADVANCED"; `complexity_guidance` is text providing specific guidance for generating outputs at the given complexity level; `is_final_step` is a boolean indicating whether this tool is the final step in the plan (affects instruction 5).

A.2 An Example of SLATE Data

This section presents an illustrative example in the SLATE format, illustrating the transformation from a user query to a hierarchical execution plan, associated tool calls, and deterministic simulation outputs. The example demonstrates the procedure structure and execution semantics used for step-wise and end-to-end evaluation.

Case

I want to create a limited-time summer promotion for my **"ThermoFlex Water Bottle"** (**SKU:** TF-WB-2023) that offers **15% off** when customers **buy 2 or more units**. The promotion should run from **June 1, 2024** to **August 31, 2024**, with a minimum purchase requirement of **$35.00**. How do I set this up and create a promo code **"SUMMERTF24"** that customers can use at checkout?

Plan

- **step**: 1. Verify product information
  - **step**: 1.1 Retrieve product details
  - **action**: `get_product_details` (**sku**="TF-WB-2023")
- **step**: 2. Create the promotion
  - **step**: 2.1 Set up promotion parameters
  - **action**: `create_promotion` (**product_id**=OUTPUT_FROM_STEP_1.1.product_id, **discount_percentage**=15, **min_quantity**=2, **min_purchase**=35.00, **start_date**="2024-06-01", **end_date**="2024-08-31")
- **step**: 3. Configure promo code
  - **step**: 3.1 Create promo code
  - **action**: `create_promo_code` (**promotion_id**=OUTPUT_FROM_STEP_2.1.promotion_id, **code**="SUMMERTF24")
- **step**: "4. Verify and activate the promotion"
  - **step**: "4.1 Validate promotion setup"
  - **action**: `validate_promotion` (**promotion_id**=OUTPUT_FROM_STEP_2.1.promotion_id)
  - **step**: "4.2 Activate the promotion"
  - **action**: `activate_promotion` (**promotion_id**=OUTPUT_FROM_STEP_2.1.promotion_id, **promo_code_id**=OUTPUT_FROM_STEP_3.1.promo_code_id)

Figure 10: Example of plan generation and tool execution.

Simulate Outputs with context

- `get_product_details` (**sku**="TF-WB-2023")
  ⇒ Output: **product_id**="P-TF-WB-2023-001"
- `create_promotion` (**product_id**="P-TF-WB-2023-001", **discount_percentage**=15, **min_quantity**=2, **min_purchase**=35.00, **start_date**="2024-06-01", **end_date**="2024-08-31")
  ⇒ Output: **promotion_id**="PROMO-TF-2024-S001"
- `create_promo_code` (**promotion_id**="PROMO-TF-2024-S001", **code**="SUMMERTF24")
  ⇒ Output: **promo_code_id**="PC-SUMMERTF24-001"
- `validate_promotion` (**promotion_id**="PROMO-TF-2024-S001")
- `activate_promotion` (**promotion_id**="PROMO-TF-2024-S001", **promo_code_id**="PC-SUMMERTF24-001")
  ⇒ Output: **success**="true"

Deterministic Simulation Dictionary

1. `get_product_details` (Args: sku)
`TF-WB-2023` ⇒ `P-TF-WB-2023-001`
Other value ⇒ different default value

2. `create_promotion` (Args: product_id, discount_percentage, min_quantity, min_purchase, start_date, end_date)
`P-TF-WB-2023-001, 15, 2, 35, 2024-06-01, 2024-08-31` ⇒ `PROMO-TF-2024-S001`
Other value ⇒ different default value

3. `create_promo_code` (Args: promotion_id, code)
`PROMO-TF-2024-S001, SUMMERTF24` ⇒ `PC-SUMMERTF24-001`
Other value ⇒ different default value

Figure 11: Deterministic simulation dictionary for tool calls.

B Experimental Details

Candidates. Every candidate list contains 52 options, one per single-character label (A–Z, then a–z), which keeps the option encoding identical across models. Two options are always present: the reference tool and a "`no_tool_needed`" option. The other 50 are retrieved with Cohere Embed v4 (`cohere.embed-v4`): library tools are ranked by the cosine similarity between the embedding of "`name: description`" and the embedding of the user query concatenated with the substep text, and the 50 highest-ranked tools other than the reference tool are kept. The list is shuffled with a seed derived from the task and substep identifiers, and the same order is shown to a decision model and to its backbone LLM; two further orders are drawn with different seeds. Because the reference tool is always included, retrieval does not cap accuracy in these experiments. Without this guarantee, the 51 top-ranked tools would contain the reference tool for 91.1% of tool substeps, and for every tool substep of a task in only 48% of tasks.

Context. The state lists the user query and every earlier substep with its reference tool and simulated output. Each earlier output is truncated to 400 characters, so that every prompt fits in a 4,096-token window.

Scoring. The prerequisites of a substep are the earlier tool steps whose outputs its reference arguments use. A correctly selected tool counts as executed only when all of its prerequisites were executed. A task succeeds when every reference outcome is reached, i.e., the recorded outputs of the final tool step and of every tool step whose output no later step uses.

Readouts. Each decision model uses its own published inference recipe (prompt format and readout) on the shared state, step, and candidates. Each backbone LLM receives the prompt below through its chat template with thinking disabled, and its prediction is the label (A–Z, a–z) with the highest next-token probability. All checkpoints are pinned to the revisions in Table 4 and run in bf16 with Hugging Face Transformers.

Table 4: Checkpoints (Hugging Face repositories and revisions). Each decision model is trained from the backbone LLM in its row.

Decision model	Backbone LLM	Readout
`Praveenrajus/jevify-smollm3-3b-apo-readout @553e4965`	`HuggingFaceTB/SmolLM3-3B-checkpoints @cfb32d50`	label probabilities (LoRA) [30]
`JevAgentHarness/AgentDecision-4B @f0e13bab`	`Qwen/Qwen3-4B @1cfa9a72`	label probabilities (LoRA) [29]
`StrandsAgents/strands-decider-E4B-gemma4-v1-2610 @1344d443`	`google/gemma-4-E4B-it @ee0ef602`	pointer head (LoRA) [27]
`Praveenrajus/Llama-3.1-Tulu-3-8B-DPO-jevify-readout @3918311a`	`allenai/Llama-3.1-Tulu-3-8B-DPO @a7beb67e`	label probabilities (LoRA) [30]
`openjev/OpenJev-Flash-9B @b77b0fff`	`Qwen/Qwen3.5-9B @c2022362`	label probabilities (merged) [28]
`StrandsAgents/strands-decider-12B-gemma4-v1-2610 @1202634a`	`google/gemma-4-12B-it @43c9d014`	pointer head (LoRA) [27]

Decision models versus LLMs: an example. A regular LLM answers by generating text, which must then be parsed into a choice. A decision model instead scores the given options and returns a probability distribution over them in one forward pass. To compare the two on equal terms, we read the backbone LLM's next-token probabilities over the option labels. Figure 12 shows one logged decision from our runs.

Error analysis. Table 5 breaks errors down by type. Nearly all errors select a wrong tool; skipping a needed tool or calling a tool for a no-tool substep is rare. Accuracy on substeps whose description names the reference tool is similar to or lower than on the other substeps. Table 6 reports results by task category.

One decision: OpenJev-Flash-9B and its backbone LLM (Qwen3.5-9B)

State. TASK: I need to add gift wrapping to my order #RT-78296431 … for the Bose QuietComfort Ultra Headphones (SKU: BQ-2023-BLK) … Can I add a personalized message and choose a holiday-themed wrapping paper before it ships tomorrow? … HISTORY: (none)
Current step. 1.1 Check if gift wrapping is available for the headphones
Options (4 of 52 shown).
B. check_gift_wrapping_availability – Check if gift wrapping is available for a specific product
S. validate_note_character_limit – Validate if requested note length is within allowed limits
p. add_gift_wrapping_to_order – Add gift wrapping and personalization options to an existing order
a. no_tool_needed – No tool call is needed for this step.

Decision model (OpenJev-Flash-9B), probabilities over the 52 options: B 0.873, Z 0.030, S 0.023, … ⇒ B (correct).
Backbone LLM (Qwen3.5-9B), next-token probabilities over the labels: p 0.444, B 0.346, b 0.127, … ⇒ p (incorrect: it adds the wrapping instead of first checking availability).

Figure 12: A decision from the experiments (task 177, step 1.1, first option order). Z and b are `generate_gift_wrapping_confirmation` and `append_express_upgrade_followups`.

Table 5: Error types and tool-step accuracy (%), pooled over three option orders. Error columns are shares of all substeps: a wrong tool selected, no tool selected for a tool step, or a tool selected for a no-tool step. The last two columns split tool-step accuracy by whether the substep text names its reference tool (16.1% of tool steps do).

		Errors (% of substeps)			Tool-step accuracy	
Model	**Type**	**Wrong tool**	**Missed tool**	**Spurious tool**	**Step names its tool**	**Other steps**
SmolLM3-3B (APO)	Backbone LLM	58.1	3.5	1.8	36.3	37.4
Jevify-SmolLM3-3B	Decision model	44.5	1.3	2.0	53.0	53.4
Qwen3-4B	Backbone LLM	31.3	1.1	1.8	66.7	66.9
AgentDecision-4B	Decision model	26.0	0.0	1.7	73.9	73.3
Gemma-4-E4B-it	Backbone LLM	33.5	0.2	1.9	53.2	67.9
Strands-Decider-E4B	Decision model	53.9	0.0	2.0	26.3	48.6
Tulu-3-8B-DPO	Backbone LLM	55.7	0.3	2.0	42.1	42.9
Jevify-Tulu-3-8B	Decision model	37.4	4.5	2.0	54.2	57.8
Qwen3.5-9B	Backbone LLM	28.2	2.5	1.7	65.1	69.4
OpenJev-Flash-9B	Decision model	22.7	0.7	1.8	73.9	76.5
Gemma-4-12B-it	Backbone LLM	22.6	2.1	0.9	73.7	75.0
Strands-Decider-12B	Decision model	41.4	0.0	2.0	40.4	61.1

Table 6: Results by task category (%), over the twelve models in Table 3 and three option orders.

User	**Category**	**Tasks**	**Substeps per task**	**Mean step-wise accuracy**	**Best step-wise accuracy**	**Best task success**
Buyer	Gifts & Discount Codes	6	11.3	66.7	83.3	22.2
Buyer	Order & Delivery Changes	7	9.9	52.4	71.0	9.5
Buyer	Pre-purchase Product Questions	11	10.1	53.8	70.6	3.0
Buyer	Product Search & Comparison	12	10.3	59.7	75.8	8.3
Buyer	Subscriptions & Returns	8	10.2	60.3	80.5	8.3
Seller	Customer Questions & Reviews	8	10.5	49.5	74.6	4.2
Seller	Fulfillment & Shipping	13	12.2	56.7	74.7	5.1
Seller	Inventory & Sales Analytics	10	10.4	59.3	73.4	6.7
Seller	Pricing Changes	11	11.5	64.6	79.3	9.1
Seller	Product Information Updates	8	10.5	58.0	80.6	16.7
Seller	Promotions	6	11.8	69.4	86.9	11.1

Prompt for the backbone LLMs

System: You are the decision module of an AI agent. Read the state and choose the best option for the current step.

User:
TASK: {user query}
HISTORY:
{substep} tool: {reference tool} result: {output, at most 400 characters}
...
CURRENT STEP: {current substep}
Which tool should the agent call for the current step?
OPTIONS:
A. {tool name} – {tool description}
...

z. {tool name} – {tool description}
Answer with the option label only.

Figure 13: Prompt for the backbone LLMs. The 52 candidates are labeled A–Z, then a–z.

C Limitations and Ethical Considerations

Limitations. SLATE is synthetic: its queries, procedures, and tool schemas are LLM-generated and its execution outcomes come from a simulator rather than live APIs. This enables reproducible, large-scale evaluation but does not capture the full behavioral complexity of real enterprise systems. Because generation used Claude models, our study evaluates non-Claude, open models to avoid rewarding generator-family bias; a sweep over larger and closed models is left to future work. Our results cover six open models up to 12B parameters evaluated on SLATE itself—we do not report cross-benchmark or cross-domain comparisons here. They isolate API selection by supplying reference arguments to correctly selected tools; argument generation is supported by the harness but not evaluated here. Some substep descriptions name their reference tool (16.1% of tool steps), although accuracy on these substeps is not higher than on the others (Table 5). The benchmark is instantiated in e-commerce; while the construction pipeline is domain-agnostic, validating difficulty in other domains (e.g., IT operations, finance) remains future work. Finally, runtime call variations are matched to recorded entries by fixed normalization rules, which may reject some valid paraphrases of free-form argument values.

Ethical considerations. SLATE contains only synthetic data and no personally identifying information. It is released for research on evaluating and improving tool-use agents; as with any automation benchmark, deployment of agents trained or selected against it should include human oversight for consequential actions.