E-BENCH: BENCHMARKING MULTI-STEP TOOL-USE AGENTS IN REAL-WORLD PRODUCT SCENARIOS

Anonymous authors
Paper under double-blind review

ABSTRACT

Large Language Models (LLMs) are increasingly deployed as agents that interact with stateful environments over multiple steps: gathering hidden information, composing tool calls, and committing state changes. We refer to this capability as **multi-step tool use**. Existing benchmarks have advanced tool-use agent evaluation, but often focus on isolated API calls, short trajectories, or settings that are difficult to scale or control. We introduce `E-Bench`, a ***fully synthetic*** benchmark with 323 state-changing tasks across three product domains: *Honor of Kings*, *QQ Music*, and *Tencent Meeting*. `E-Bench` decouples *environment synthesis* from *task synthesis*: graph-guided database filling builds reusable, orphan-free product environments, while generator-solver asymmetry creates tasks with both an *information gap* and a *tool gap*, requiring agents to discover hidden data and compose multiple tool calls before changing state. Outcomes are graded deterministically by database-state diffs. Since both environments and tasks are synthetic, `E-Bench` is controllable at the environment level and scalable at the task level. Benchmarking 11 cutting-edge LLMs shows that multi-step tool use remains challenging: Pass^3 stays below 60% for the strongest models, and even with code execution in `E-Bench`-Code extension, reliability (Pass^3) remains below 70%.

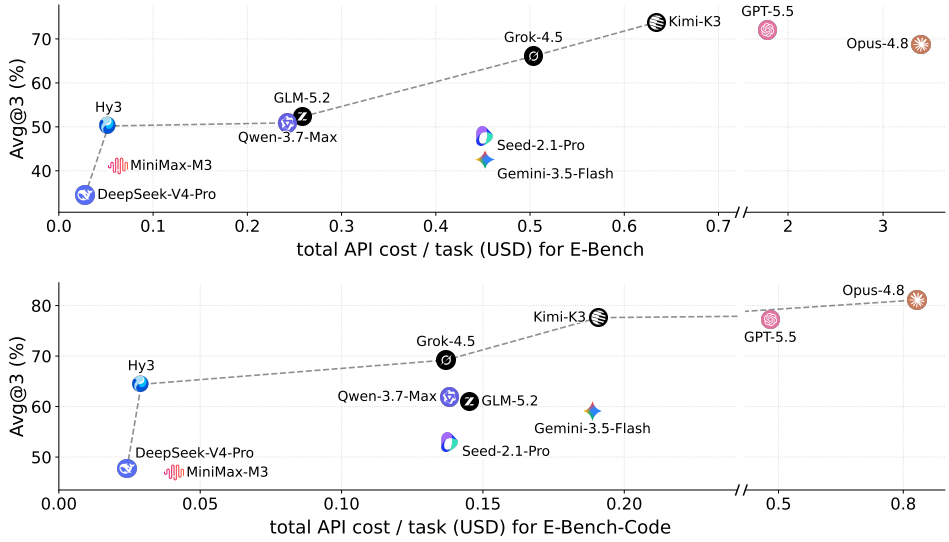
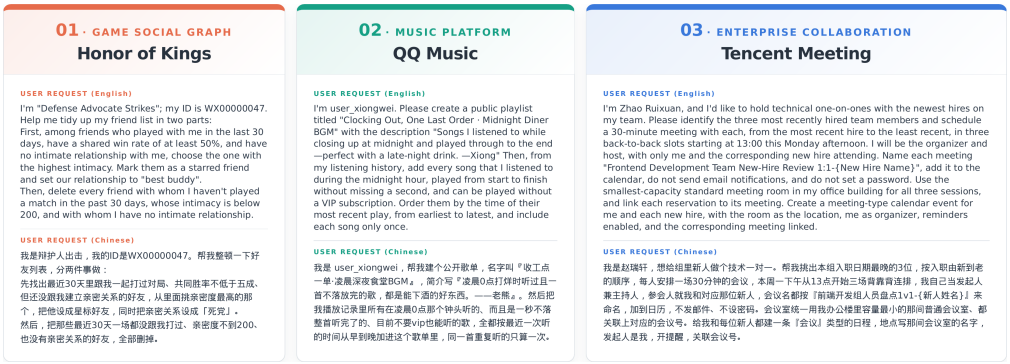


Figure 1: Avg@3 (%) versus total API cost per task (USD) under `E-Bench` (top) and `E-Bench`-Code (bottom). Calculation details are included in the Appendix C.4. The dashed line marks the Pareto (cost-efficiency) frontier.

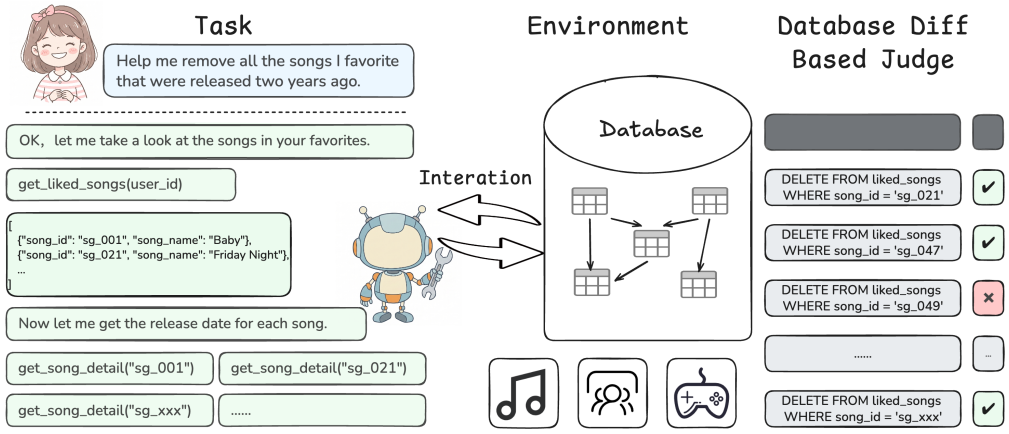
1 INTRODUCTION

Currently, Large Language Models (LLMs) have moved beyond answering self-contained prompts and are increasingly deployed as agents that interact with external environments to accomplish complex tasks Yao et al. (2022); Patil et al. (2024); Singh et al. (2025); Anthropic (2024). Success then requires more than a plausible response: an LLM agent must *repeatedly* identify missing

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.



(a) Representative natural-language tasks across the three domains.



(b) Tool-mediated task execution and deterministic database-diff evaluation.

Figure 2: Overview of E−Bench. (a) Natural-language tasks span diverse product domains. (b) Agents use domain-specific tools in fully-synthetic stateful environments, and are evaluated by comparing database-state diffs to ground truth.

information, decide which tools to invoke, integrate observations across steps, and commit changes back to a stateful environment (we term this **multi-step tool use**). This shift from answering to acting calls for benchmarks that evaluate agentic behavior under realistic interaction constraints, rather than isolated language understanding or single-step tool selection.

This paradigm underlies many of the most valuable real-world applications, from operating software and querying databases to orchestrating business workflows, yet such environment-grounded agency remains hard to evaluate systematically. Recent benchmarks probe models' ability to invoke APIs correctly Patil et al. (2024); Qin et al. (2024); Patil et al. (2025) and to operate in realistic web, OS, and software-engineering environments Zhou et al. (2024); Trivedi et al. (2024); Jimenez et al. (2024). While these reveal weaknesses in planning, tool grounding, and long-horizon execution, they center on short tool-use trajectories Yao et al. (2024); Mialon et al. (2023), isolated API calls Patil et al. (2024; 2025), or static QA Rein et al. (2024); Hendrycks et al. (2021); Yang et al. (2018) without environment modification, and thus fail to capture agentic capability under partial observability, heterogeneous tools, long-horizon dependencies, and precise state-changing operations.

Moreover, although benchmarks built on real systems offer rich, stateful interactions, they are hard to scale, expensive to annotate, sensitive to data-safety and privacy constraints, and unstable to reproduce as the underlying services evolve Yao et al. (2024); Liu et al. (2024); Xu et al. (2024). They can also be difficult to decontaminate: when tasks depend on public facts, familiar interfaces, or recurring workflows, model performance may reflect prior exposure rather than the ability to actively acquire hidden state through tools. A faithful benchmark must therefore present a complete, controllable environment in which the model cannot shortcut the intended reasoning, while remaining cheap to scale and safe to release. To this end, we introduce E−Bench, a ***fully synthetic*** benchmark for evaluating multi-step tool-use agents. E−Bench comprises 323 state-changing tasks spanning three domains modeled after real products: *Honor of Kings*, *QQ Music*, and *Tencent Meeting*. As illustrated in Figure 2, these tasks take the form of realistic natural-language requests (Figure 2a) and

are solved through tool-mediated interaction with a stateful product environment, with correctness determined by exact database-state differences (Figure 2b).

E-Bench is built in two decoupled stages: *environment synthesis* and *task synthesis*. For each domain, we first construct a reusable, fully mocked product environment rather than task-specific state snapshots, and then synthesize many state-changing tasks from this shared environment. In the first stage, we synthesize a large, diverse, and consistent database via *graph-guided database filling*. Specifically, we construct a table-dependency graph from the relational schema and use a powerful LLM to populate tables in topological order, enforcing referential integrity by construction. This yields an orphan-free, stateful product world rather than tool stubs or task-local fixtures. In the second stage, a privileged task *generator* with SQL and code access explores this environment and authors tasks through a realistic product loop: *inspect data* $\rightarrow$ *decide targets* $\rightarrow$ *modify state* $\rightarrow$ *summarize intent*. For each task, it records the induced database change as a ground-truth diff and annotates the exercised tool-use capabilities. Since tasks are derived from a shared reusable environment, new tasks can be generated by varying target entities, constraints, and reasoning patterns without redesigning the underlying data. Moreover, because all tasks share a complete underlying environment rather than task-specific, incomplete local snapshots, solvers have greater room to explore the available context, better reflecting the contextual richness encountered in real-world scenarios.

During benchmarking, the agent (the *solver*) has no direct database or SQL access. In the base E-Bench setting, it is also denied code execution, while E-Bench-Code grants this capability. This generator-solver asymmetry creates an *information gap* and a *tool gap*: solvers must discover hidden data, reason from partial observations, and compose available tools (e.g., issuing parallel independent calls) to produce the correct state change. Outcomes are graded deterministically by comparing the final database state with the ground-truth diff, without any LLM judge.

By evaluating agents through multi-step tool interaction with simulated product environments, E-Bench measures their ability to gather necessary information, coordinate tool calls, and decide what to retrieve before acting. Since both environments and tasks are synthetically constructed rather than drawn from real users or live services, E-Bench is controllable at the environment level and scalable at the task level for training-oriented use. Our contributions are as follows:

(1) We introduce E-Bench, a fully synthetic benchmark for systematic evaluation of LLM agents on *multi-step tool use*, which comprises 323 state-changing tasks across three domains modeled after real corporate products.

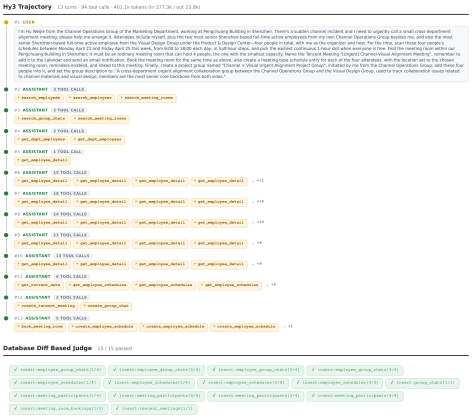
(2) We design a fully synthetic construction pipeline that decouples *environment synthesis* from *task synthesis*, enabling a single reusable, integrity-preserving environment to support scalable task generation. Evaluation uses deterministic database-state diffs rather than LLM judges.

(3) We benchmark 11 cutting-edge LLMs as tool-use agents and find that multi-step tool use remains far from solved: the strongest model (Kimi-K3) reaches only 73.8% Avg@3, while consistency-oriented reliability, measured by Pass^3, remains below 60%. Granting code execution in E-Bench-Code improves accuracy, but performance consistency remains limited with Pass^3 below 70%.

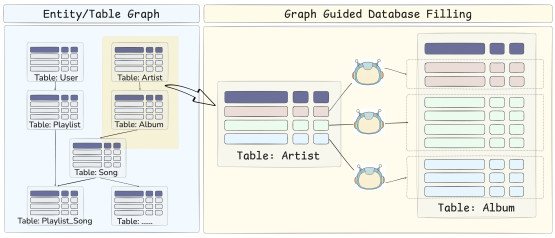
2 E-BENCH: A BENCHMARK FOR MULTI-STEP TOOL USE

2.1 OVERVIEW

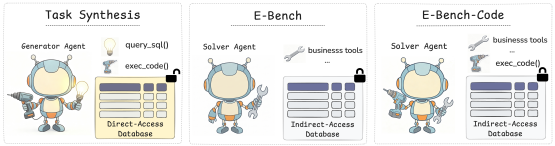
E-Bench follows one principle: evaluating multi-step tool use requires both a coherent environment and tasks that cannot be solved without interacting with it. To this end, E-Bench explicitly decouples *environment synthesis* from *task synthesis* in two stages. First, we construct a ***purely synthetic product environment*** from a relational schema and populate it through ***graph-guided database filling***, yielding a coherent, orphan-free product world rather than isolated tool stubs or task-local fixtures. Second, we synthesize state-changing tasks over this reusable environment through a controlled ***generator-solver asymmetry*** that introduces ***an information gap and a tool gap*** by design, forcing solvers to recover hidden state, compose multi-step and often parallel tool calls, and execute verifiable state changes. Because tasks are derived from a shared, pre-populated environment rather than hand-authored fixtures, task construction is scalable. E-Bench instantiates this pipeline across three domains—*Honor of Kings*, *QQ Music*, and *Tencent Meeting*—producing



(a) A representative Hy3 trajectory on a *Tencent Meeting* task, coordinating participant selection, schedule search, room reservation, meeting creation, and group setup.



(b) Graph-guided database filling with constrained context-aware synthesis for reliable environment construction.



(c) The *information gap* and *tool gap* between *generator* and *solver* for scalable task construction.

Figure 3: Overview of E−Bench evaluation trajectory ((a)) and construction ((b) and (c)).

323 state-changing tasks. We show one task example from *Tencent Meeting* in Figure 3a with more included in the Appendix D.

The remainder of this section is organized as follows. Section 2.2 details graph-guided environment construction; Section 2.3 describes automatic task construction and deterministic evaluation; and Section 2.4 introduces E−Bench-Code, which grants solvers code execution to partially close the tool gap. Together, these designs enable E−Bench to test ***full-information acquisition and multi-step parallel tool use*** under realistic product-state feedback. In total, E−Bench spans three reusable product environments and 323 state-changing tasks; comprehensive per-domain statistics and benchmark composition are reported in Appendix B.

2.2 Reliable Environment Construction

E−Bench first constructs, for each domain, a product environment fully simulated by a database. Each domain is defined by a set of tables, where each table corresponds to a concrete entity or relation (e.g., users, artists, albums, songs, and playlists in the music domain) and carries fields describing entity attributes (e.g., an album's release date or a meeting's start time and room location). The goal of environment construction is therefore not to generate a small set of plausible examples, but to build a product-level data world with diverse entity types, multi-hop relations, and historical states.

Graph-Guided Database Filling. Tables within each domain are linked by explicit foreign-key dependencies: for example, albums reference existing artists, songs reference existing albums, and playlist-song relations reference both existing playlists and songs. These dependencies determine record validity, since a row is meaningful only when its referenced entities exist and are semantically consistent. E−Bench therefore converts each relational schema into a table-level dependency graph (see also the left panel in Figure 3b), initializes an empty database with primary-key and foreign-key constraints, enables foreign-key checking, and populates tables in topological order. Root tables such as those for artists, departments, and meeting rooms are generated first, followed by downstream tables only after their dependencies are materialized. This process guarantees referential integrity by construction rather than repairing violations after free-form generation.

Constrained Context-Aware Synthesis. During database synthesis, E−Bench uses a powerful LLM as a constrained synthesizer rather than asking it to generate an entire database freely. When populating a table, the model receives the table schema, key annotations, generation requirements, global domain context, and relevant existing records. For tables dependent on upstream entities, it generates child records conditioned on each parent, like shown in Figure 3b. For example, when creating playlist-song relations, the model observes the current playlist and sampled candidate songs, and may issue read-only queries if additional candidates are needed. This process is constrained by tool-mediated interaction with the database. The model inserts records only through insertion tools, allowing the database to enforce primary-key uniqueness and foreign-key validity at insertion time. For downstream tables that must select from existing objects, the model is forced to query candidate

entities from the current database state rather than inventing identifiers. The system also samples relevant records and retrieves parent-related context along the dependency graph, keeping prompts compact while preserving cross-table consistency.

Post-Generation Validation and Repair. After database population, E-Bench applies deterministic validation and repair scripts to ensure the fidelity and integrity of the constructed environment. These scripts correct semantic errors that may remain in LLM-synthesized data, such as inconsistent temporal ordering, incorrect aggregate counts, or mismatched status fields, by verifying and automatically repairing cross-field, cross-table, and temporal constraints, improving the consistency and quality of the final environment.

Domain-Specific Tool Construction. After validation, E-Bench builds domain-specific MCP tools on top of the database. These tools function as CRUD interfaces over the underlying database, but expose product-level semantics to agents, such as searching songs, viewing playlists, adding favorites, querying meeting rooms, creating meetings, and updating schedules. Because all tools operate over the same pre-populated, referentially complete database, agents receive realistic product feedback during interaction. We further validate the implementations of these MCP tools with unit tests to ensure that tool behavior remains consistent with database semantics.

Overall, this pipeline yields a structurally complete, cross-table consistent, and tool-interactive product state space that, unlike task-coupled benchmarks, contains no orphan records and provides a stable foundation for automatic task synthesis.

2.3 Scalable Task Construction

The second stage turns each environment into benchmark tasks under one principle: a task is worth including only if solving it genuinely requires interacting with the environment. E-Bench enforces this through a controlled asymmetry between task *generation* and *solving*: during generation, a strong *task generator* (Claude Opus 4.7) works on a temporary database copy and may inspect global state and run code, whereas at evaluation the *solver* sees only a natural-language request and the public domain tools. This lets E-Bench author tasks that are deterministically solvable yet cannot be shortcut, forcing agents to recover hidden information and induce the required state change through ordinary tool calls. The asymmetry has two faces—an *information gap* and a *tool gap*, introduced by construction—which we describe next (see also Figure 3c).

The Information Gap. The information gap withholds global database state from the solver, so a task cannot be answered from its prompt alone. Seeing the full schema and read-only `query_sql`, the generator can define tasks whose targets depend on latent state rather than named values—e.g., all songs meeting a hidden condition, the intersection of two sets, the top entity under an aggregation, entities several foreign-key hops away, or the earliest feasible meeting slot. An intent-rewriting step then keeps this knowledge hidden, so the query says "add the unavailable songs in my favorites" rather than listing song IDs or their count (details in Appendix B.2). The gap thus targets *full-information acquisition*: it tests whether an agent can actively gather the *complete* hidden target set before acting, rather than commit prematurely on partial evidence.

The Tool Gap. The tool gap makes the generator computationally stronger than the base solver. During task generation, the generator may invoke internal tools such as `query_sql` and `exec_code`, enabling programs that paginate results, compute aggregates, perform set operations, traverse multi-hop relations, and chain dependent writes, and thereby author tasks whose ground truth requires genuine computation rather than a single domain-specific tool call. At evaluation time, internal tools are removed, leaving agents with only business-level domain-specific MCP tools; the solver must therefore reproduce the target outcome through ordinary tool calls. This gap tests whether agents can correctly compose and order multi-step tool use—including issuing independent *parallel tool calls* when permitted by the environment—to perform computation they cannot directly offload. E-Bench-Code partially closes this gap by granting solvers code execution, and comparing the two settings quantifies how much code execution raises the performance ceiling.

Capability Taxonomy as Synthesis Guidance. With the two gaps, we further define six capability types for generated tasks as listed in Table 3 in Appendix B. These capabilities arise naturally from our benchmark design and fall into two broad groups. The first group, namely *Full-Data Acquisition*, *Multi-Condition Filtering*, and *Aggregation and Computation*, is driven primarily by the information

Table 1: Model performance on E-Bench and E-Bench-Code. Models are ranked by E-Bench Avg@3. The highest three values in each column are highlighted in **bold**, underline and gray background.

	Model	E-Bench			E-Bench-Code		
		Avg@3	Pass@3	Pass^3	Avg@3	Pass@3	Pass^3
1	Kimi-K3	**73.79%**	**87.62%**	**58.82%**	77.61%	88.70%	65.80%
2	GPT-5.5	72.03%	82.97%	57.59%	77.19%	87.32%	66.90%
3	Opus-4.8	68.78%	84.33%	50.81%	**81.11%**	**92.61%**	**68.66%**
4	Grok-4.5	66.10%	80.50%	52.32%	69.24%	82.66%	55.75%
5	GLM-5.2	52.32%	71.52%	30.96%	60.99%	81.69%	42.25%
6	Qwen-3.7-Max	50.88%	70.59%	30.34%	61.92%	80.99%	44.01%
7	Hy3	50.15%	74.30%	26.63%	64.40%	85.56%	44.01%
8	Seed-2.1-Pro	47.94%	65.33%	28.79%	53.04%	75.35%	30.99%
9	Gemini-3.5-Flash	42.62%	64.71%	21.98%	62.54%	82.39%	40.49%
10	MiniMax-M3	41.07%	62.85%	20.12%	46.85%	74.65%	21.13%
11	DeepSeek-V4-Pro	34.47%	53.56%	17.34%	47.68%	72.89%	25.35%

gap: agents must retrieve, aggregate, and compute over large volumes of hidden data. The tool gap further amplifies the difficulty of these tasks, but closing it can partially alleviate the burden. The second group, *Cross-Step Dependency*, *Precise Boundary Judgment*, and *Cross-Entity Cascade*, on the other hand, places greater emphasis on reasoning and decision making, requiring agents to determine the next action from the information accumulated so far.

Task Generation and Validation. Each task is generated through a product-oriented loop—*inspect data → decide targets → modify state → summarize intent*—for a sampled user and capability type(s). We snapshot the database before and after the generator's writes and record the exact state diff as ground truth. Surviving tasks are then validated by three strong *validators* given the same full-information tools as the generator, and are accepted only if at least two validators reproduce changes consistent with the ground truth and a weak baseline does not solve them too easily. Because correctness is checked against deterministic database-state diffs rather than an LLM judge, evaluation is stable and less sensitive to semantic-judging variance. We detail the intent-rewriting, validation chain, and task specification format in Appendix B.2.

2.4 E-BENCH-CODE: CLOSING THE TOOL GAP

Because E-Bench withholds the code-execution capability from the solver, a *tool gap* remains between the strong task generator and the solver (Section 2.3). Our extension E-Bench-Code partially closes it by granting the solver the generator's `exec_code` tool, with which the agent writes Python that calls the domain-specific MCP tools as inner-functions—paginating, aggregating, and performing set operations programmatically instead of manually orchestrating long call sequences. All else is held fixed (tasks, tools, database copies, and the state-diff evaluator), so any performance difference is attributable to this interface.

Importantly, E-Bench-Code closes the *tool gap* but not the *information gap*: `exec_code` exposes only the business-level domain-specific APIs, not `query_sql` or raw database access, so agents must still discover hidden targets through the observable environment. The two settings thus isolate complementary abilities—manual coordination of long tool-call sequences versus reasoning about what to retrieve (see Appendix B.2).

3 MAIN RESULTS

3.1 EVALUATION SETUP

Models. We evaluate 11 frontier LLMs, selecting the newest and strongest model from each developer at the time of writing: GPT-5.5, Opus-4.8[1], Grok-4.5, GLM-5.2, Qwen-3.7-Max, Hy3, Seed-2.1-Pro, Gemini-3.5-Flash, MiniMax-M3, Kimi-K3, DeepSeek-V4-Pro and Muse-Spark-1.1[2]. Each model is run as an agent with their highest available thinking effort using a shared harness which exposes each domain's MCP tools and manages the multi-turn interaction loop.

[1]We use Claude Opus 4.7 as *generator* and test a different model Opus-4.8 as *solvesor*.

[2]Due to Meta's strict content moderation, Muse-Spark-1.1 results are reported only in Appendix C.5, Table 6.

Experiment and metrics. Each task is attempted in three independent trials, each starting from a fresh, isolated copy of the domain database. A trial succeeds only if the agent's final database state exactly matches the ground-truth diff under our deterministic rule-based verifier, with no partial credit. We report three metrics of increasing strictness: *Avg@3*, the per-trial success rate averaged over all tasks and trials; *Pass@3*, counting a task solved if at least one trial succeeds; and *Pass*^{3}, requiring all three trials to succeed, which measures reliability. Beyond accuracy, we record the number of tool calls, agent action turns, and input/output tokens averaged across tasks to characterize interaction cost. We evaluate all models under both the base `E-Bench`, where solvers access only domain-specific MCP tools, and `E-Bench`-Code, which additionally provides `exec_code`.

3.2 Main Results

Table 1 reports the performance of all evaluated models under `E-Bench` and `E-Bench`-Code, ranked by Avg@3, while Figures 4 and 5 compare the per-domain Avg@3 and interaction cost (MCP tool calls and agent turns) under the two settings.

3.2.1 E-Bench

Multi-step tool use remains far from solved. Even the strongest agents leave substantial headroom on `E-Bench`. Kimi-K3 achieves the highest Avg@3 at 73.79%, followed by GPT-5.5, Opus-4.8, and Grok-4.5, all of which exceed 66%, while no other model exceeds 53%. Also, across all 11 models, Avg@3 averages only 54.56%. This indicates that a typical agent fails on nearly half of the tasks in a single attempt.

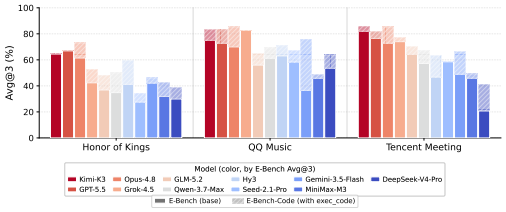


Figure 4: Per-domain Avg@3 (%) for the 11 evaluated models. Ordered by `E-Bench` Avg@3.

Reliability remains the central bottleneck. The gap between Pass@3, Avg@3, and Pass^3 shows that many agents succeed only intermittently, with substantially higher Pass@3 compared to Pass^3. Even top models are non-robust: Kimi-K3 drops from 87.62% Pass@3 to 58.82% Pass^3, and GPT-5.5 drops from 82.97% to 57.59%. For weaker models, reliability nearly collapses. As `E-Bench` requires exact state changes with no partial credit, such instability is practically significant: agents that solve tasks only occasionally are not yet dependable for modifying live product state.

3.2.2 E-Bench-Code

Code execution raises the performance of all models, but reliability remains limited. Granting agents `exec_code` in `E-Bench`-Code improves Avg@3 for all evaluated models, whether the model is strong or not. The gains are substantial: Opus-4.8 rises from 68.78% to 81.11%, taking the top spot over Kimi-K3 and GPT-5.5; Gemini-3.5-Flash climbs from 42.62% to 62.54%—a striking 46.7% relative improvement; weaker models like DeepSeek-V4-Pro also gain a 38.32% relative improvement. However, even with code execution, the best Pass^3 (Opus-4.8) remains below 70% and the worst is only 21.13%, leaving substantial room for reliability improvement.

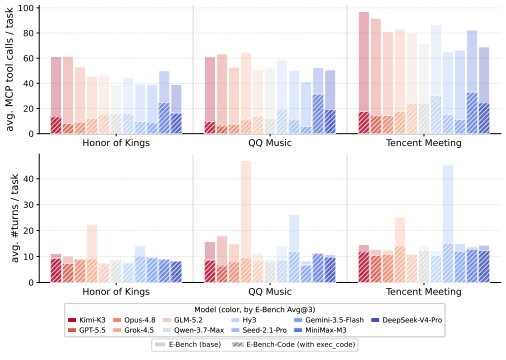


Figure 5: Per-domain MCP tool calls (top row) and number of turns (bottom row) per task for the 11 evaluated models. Ordered by `E-Bench` Avg@3.

Code execution reshuffles the leaderboard benefiting stronger code users. Models whose rankings rise on the leaderboard tend to be those that exploit `exec_code` effectively: Gemini-3.5-Flash from 9th to 6th, while Opus-4.8 (a known strong coder) takes the top position instead of the third place it holds in `E-Bench`. These models route a large fraction of their interaction through code (e.g., 95% for Gemini-3.5-Flash and 93% for Opus-4.8), replacing many individual tool calls with compact programs (i.e., often calling domain-

specific functions inside `exec_code`, rather than direct calling their corresponding MCP tools). We analyze this behavior further in Section 3.5.

Code execution improves accuracy while cutting cost. The performance gains from `exec_code` come with markedly *lower* interaction cost. As shown in Figure 5, enabling `exec_code` reduces the average number of MCP tool calls per task from 60.42 to 15.86 (a 73.8% drop) and agent turns from 14.87 to 9.87 (a 33.6% drop), averaged over 11 models and three domains, as a single `exec_code` block folds multi-turn domain-function calls into one top-level action. Thus, with `exec_code`, models solve more tasks while reaching solutions at lower cost.

3.3 How Does Difficulty Vary Across Domains? Per-domain Analysis

We further analyze each domain separately with Figure 4 plotting per-domain per-model Avg@3.

Domain difficulty and code-execution gains are both uneven. Figure 4 shows that *Honor of Kings* is the most challenging domain, while *QQ Music* is the easiest. Under `E-Bench`, average Avg@3 is 43.64%, 58.87%, and 61.39% for *Honor of Kings*, *Tencent Meeting*, and *QQ Music*, respectively. With `E-Bench`-Code, these rise to 52.36%, 67.78%, and 71.94%. *QQ Music* benefits most from code execution, gaining 10.55 points on average, compared with 8.72 and 8.91 points for *Honor of Kings* and *Tencent Meeting*. This matches its capability mix: *QQ Music* contains more tasks requiring retrieval and aggregation of large amounts of data (e.g. *Full-Data Acquisition* and *Aggregation and Computation* tasks), and fewer tasks requiring reasoning for decision making (e.g. *Precise Boundary Judgment* tasks), which `exec_code` can accelerate (Section 3.4).

Most model rankings vary across domains. Across domains and settings, Kimi-K3, GPT-5.5, and Opus-4.8 form the frontier tier, with Grok-4.5 as a strong second tier. Rankings are nonetheless highly domain-sensitive—Grok-4.5, for instance, is the best base-`E-Bench` model on *QQ Music* but only mid-pack on *Honor of Kings*—and below the frontier models cluster closely and reorder substantially across domains and settings, as `exec_code` disproportionately benefits strong code users. These results show that robust multi-step tool-use evaluation requires coverage across diverse domains rather than a single domain.

3.4 Where Does Code Execution Help? Per-Capability Analysis

To further identify how `exec_code` helps, Table 2 decomposes the Avg@3 gain from adding `exec_code` by capability, averaged over the 11 evaluated models.

Table 2: Per-capability Avg@3 averaged over 11 evaluated models under `E-Bench` and `E-Bench`-Code, as well as the absolute and relative gain of granting coding capability ("Δ Avg@3" and "Rel. Δ Avg@3"). Rows are sorted by gains (Δ Avg@3). The highest value in each column is highlighted in **bold**, and the second-highest is underlined.

Capability	`E-Bench`	`E-Bench`-Code	Δ Avg@3	Rel. Δ Avg@3
Multi-Condition Filtering	51.94%	62.55%	**+10.61%**	**+20.43%**
Full-Data Acquisition	53.55%	63.65%	<u>+10.11%</u>	<u>+18.88%</u>
Aggregation and Computation	52.85%	62.30%	+9.45%	+17.88%
Cross-Entity Cascade	**60.08%**	**68.88%**	+8.80%	+14.66%
Cross-Step Dependency	54.11%	62.79%	+8.68%	+16.05%
Precise Boundary Judgment	<u>57.39%</u>	<u>64.94%</u>	+7.55%	+13.16%

Code execution helps computation more than reasoning. The largest gains from `exec_code` appear in mechanically intensive capabilities (see Table 2): *Multi-Condition Filtering*, *Full-Data Acquisition*, and *Aggregation and Computation*, with highest absolute and relative Avg@3 improvements at the same time. These tasks require exhaustive retrieval, exact filtering, aggregation, and intermediate-result tracking, which code can handle reliably through loops and iterations. In contrast, gains are smaller for *Cross-Step Dependency* and *Precise Boundary Judgment*, as these capabilities depend more on selecting the right entities, dependencies, thresholds, or action order. Thus, `exec_code` primarily offloads computation rather than reasoning and decision making, explaining both its large average gains and the remaining headroom in `E-Bench`-Code.

3.5 When Do Models Choose to Code? Per-Task Analysis

Another interesting observation is that, even when `exec_code` is available under `E-Bench`-Code, many models still solve some of the tasks without using it, either consistently or sporadically across trials. Figure 6 groups tasks by `exec_code` usage across three trials: all trials use it (all-C), no trials

use it (no-C), or usage is mixed (partial). For these groups, we compare task-averaged performance (Avg@3), MCP tool calls, and agent action turns, with paired E-Bench results as references. For clarity, we show results for three representative models here and others in the Appendix C.2.

Code execution rescues hard tasks while reducing interaction cost. Tasks where models consistently avoid exec_code (no-C) are relatively easy, often requiring the fewest MCP tool calls and shortest interaction turns. Accordingly, the two settings behave nearly identically on no-C tasks: tool calls and turns remain essentially unchanged, and Avg@3 fluctuates only slightly. In contrast, tasks where exec_code is consistently used (all-C) are the most demanding, and exec_code yields large performance gains while substantially reducing both MCP calls and agent turns, bringing costs close to those of no-C tasks. Mixed-usage tasks show the same trend: Avg@3 improves from 52.53% to 57.70%, while MCP calls drop from 32.68 to 24.34 on average. Overall, exec_code is used selectively on demanding, tool-heavy tasks, where it batches multi-step, often parallel, domain-specific tool calls into shorter execution traces controlled by more precise code.

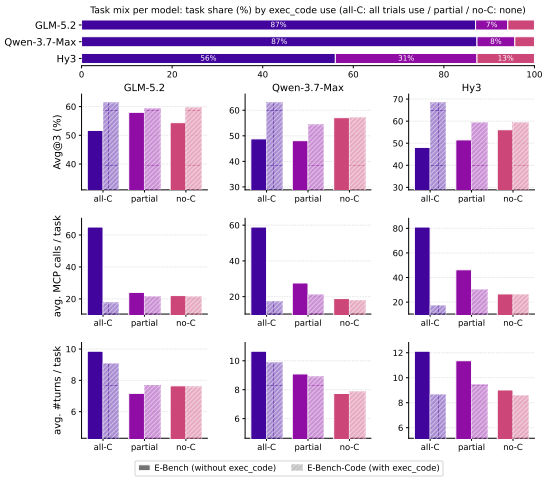


Figure 6: Comparison of task outcomes under E-Bench-Code, grouped by whether *all* or *none* trials used exec_code (all-C and no-C), or usage was mixed across trials (partial). Results are first averaged across trials and then across tasks.

3.6 Does Spending More Help? Performance versus Interaction Cost

A natural question is whether agents that fail simply need to *try harder*—issue more tool calls or consume more context. Appendix C.3 (Figure 9) plots Avg@3 against three per-task cost measures (MCP tool calls, turns, and total tokens), one point per <model, domain>.

Harder tasks require more interaction, but code execution decouples cost from work. Under E-Bench, Avg@3 is moderately positively correlated with the number of tool calls across the 33 <model,domain> pairs (Pearson $r = +0.57$; up to $+0.89$ within a single domain), matching the intended *information gap*: solvers must gather hidden information through multi-step tool use before acting, so a low call count often signals premature action rather than efficiency. Under E-Bench-Code, this correlation turns weakly *negative* ($r = -0.36$, and -0.87 in *QQ Music*): because models fold many domain-specific calls into a single exec_code block, fewer emitted calls do not imply less work, and code execution can lower cost while preserving or improving performance. More detailed analyses of efficiency and API cost (Figure 1) are provided in Appendix C and Appendix C.3.

4 Conclusion and Future Work

In this work, we introduce E-Bench and its code-enabled extension E-Bench-Code, a fully synthetic benchmark for evaluating LLM agents on *multi-step tool use*. E-Bench separates environment synthesis from task synthesis: graph-guided database filling builds reusable, orphan-free environments, while generator-solver asymmetry creates state-changing tasks with information and tool gaps that force agents to discover hidden data and compose multiple, often parallel, tool calls before changing state, all graded deterministically via database-state diffs. Benchmarking 11 frontier LLMs shows multi-step tool use remains unsolved: the best model reaches only 73.79% Avg@3 with Pass^3 below 60%, and although exec_code improves every model, the best Pass^3 stays below 70%, with gains concentrated on computation-heavy capabilities rather than reasoning-heavy ones.

Future work will extend E-Bench toward real domain CLIs and live product back-ends, and from single-domain tasks to cross-domain scenarios that better reflect real product workflows. We also view E-Bench as a controllable, scalable source of training data for improving agents' multi-step tool-use ability and reliability.

AI USE STATEMENT

In this work, we used generative AI tools for *generating synthetic data sets, designing or providing feedback on research methodology or experiments, implementing methods, drafting and polishing the paper, supporting qualitative and thematic data analysis, and interpreting results*.

We have not used generative AI tools for *assisting with translation, or cleaning and reformating dataset*, and *helping develop theoretical models or conceptual frameworks, formulating mathematical claims, providing critical ingredients for proving mathematical claims, assisting in the writing of proofs, and proposing or refining hypotheses* are not applicable to this work.

Additionally, we used generative AI tools for *creating or modifying scientific figures or images, creating or editong code, drafting parts of this research paper, summarizing and analysing existing literature, brainstorming, sourcing/searching for information, editing this research paper to improve readability, identifying relevant literature, formating references, and proposing subsection titles for this research paper*.

We have reviewed all AI-assisted work. We checked LLM-generated codes as well as its proposed paper polishing suggestions carefully by at least 2 authors. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

Both the product environments and the tasks are entirely synthesized rather than drawn from real users or live services, so the work involves no human subjects, no personally identifiable information, and no private or proprietary data, and therefore poses no privacy, data-licensing, or consent concerns. After checking the data carefully by human, we believe this work does not raise issues regarding discrimination, bias, fairness, or other items in the ICLR Code of Ethics.

REFERENCES

Anthropic. The claude 3 model family: A new standard for intelligence, 2024. URL https://www.anthropic.com/news/claude-3-family.

Chaithanya Bandi, Ben Hertzberg, Geobio Boo, Tejas Polakam, Jeff Da, Sami Hassaan, Manasi Sharma, Andrew Park, Ernesto Hernandez, Dan Rambado, Ivan Salazar, Rafael Cruz, Chetan Rane, Ben Levin, Brad Kenstler, and Bing Liu. MCP-atlas: A large-scale benchmark for tool-use competency with real MCP servers. *arXiv preprint arXiv:2602.00933*, 2026. URL https://arxiv.org/abs/2602.00933.

Wei He, Yueqing Sun, Hongyan Hao, Xueyuan Hao, Zhikang Xia, Qi Gu, Chengcheng Han, Dengchang Zhao, Hui Su, Kefeng Zhang, Man Gao, Xi Su, Xiaodong Cai, Xunliang Cai, Yu Yang, and Yunke Zhao. VitaBench: Benchmarking LLM agents with versatile interactive tasks in real-world applications. *arXiv preprint arXiv:2509.26490*, 2025. URL https://arxiv.org/abs/2509.26490.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=d7KBjmI3GmQ.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. Agentbench: Evaluating llms as agents. In *International Conference on Learning Representations*, 2024. URL https://arxiv.org/pdf/2308.03688.

Zhiwei Liu, Jielin Qiu, Shiyu Wang, Jianguo Zhang, Zuxin Liu, Roshan Ram, Haolin Chen, Weiran Yao, Shelby Heinecke, Silvio Savarese, Huan Wang, and Caiming Xiong. MCPEval: Automatic MCP-based deep evaluation for AI agent models. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 373–402, Suzhou, China, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.emnlp-demos.27. URL https://aclanthology.org/2025.emnlp-demos.27/.

Jiarui Lu, Thomas Holleis, Yizhe Zhang, Bernhard Aumayer, Feng Nan, Haoping 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. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 1160–1183. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.findings-naacl.65. URL https://aclanthology.org/2025.findings-naacl.65/.

Ziyang Luo, Zhiqi Shen, Wenzhuo Yang, Zirui Zhao, Prathyusha Jwalapuram, Amrita Saha, Doyen Sahoo, Silvio Savarese, Caiming Xiong, and Junnan Li. MCP-Universe: Benchmarking large language models with real-world Model Context Protocol servers. *arXiv preprint arXiv:2508.14704*, 2025. URL https://arxiv.org/abs/2508.14704.

Grégoire Mialon, Clémentine Fourrier, Thomas Wolf, Yann LeCun, and Thomas Scialom. Gaia: a benchmark for general ai assistants. In *The Twelfth International Conference on Learning Representations*, 2023. URL https://arxiv.org/pdf/2311.12983.

Guozhao Mo, Wenliang Zhong, Jiawei Chen, Qianhao Yuan, Xuanang Chen, Yaojie Lu, Hongyu Lin, Ben He, Xianpei Han, and Le Sun. LiveMCPBench: Can agents navigate an ocean of MCP tools? *arXiv preprint arXiv:2508.01780*, 2025. URL https://arxiv.org/abs/2508.01780.

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. URL https://arxiv.org/pdf/2305.15334.

Shishir G Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. The berkeley function calling leaderboard (BFCL): From tool use to agentic evaluation of large language models. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=2GmDdhBdDk.

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 *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=dHng2O0Jjr.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R. Bowman. GPQA: A graduate-level google-proof q&a benchmark. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=Ti67584b98.

Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, et al. Openai gpt-5 system card. *arXiv preprint arXiv:2601.03267*, 2025.

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 (Volume 1: Long Papers)*, pp. 16022–16076, 2024. URL https://arxiv.org/pdf/2407.18901.

Zhenting Wang, Qi Chang, Hemani Patel, Shashank Biju, Cheng-En Wu, Quan Liu, Aolin Ding, Alireza Rezazadeh, Ankit Shah, Yujia Bao, and Eugene Siow. MCP-bench: Benchmarking tool-using LLM agents with complex real-world tasks via MCP servers. In *International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=fe8mzHwMxN.

Zijian Wu, Xiangyan Liu, Xinyuan Zhang, Lingjun Chen, Fanqing Meng, Lingxiao Du, Yiran Zhao, Fanshi Zhang, Yaoqi Ye, Jiawei Wang, Zirui Wang, Jinjie Ni, Yufan Yang, Arvin Xu, and Michael Qizhe Shieh. MCPMark: A benchmark for stress-testing realistic and comprehensive MCP use. In *International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=uobROwBsJm.

Frank F Xu, Yufan Song, Boxuan Li, Yuxuan Tang, Kritanjali Jain, Mengxue Bao, Zora Z Wang, Xuhui Zhou, Zhitong Guo, Murong Cao, et al. Theagentcompany: benchmarking llm agents on consequential real world tasks. *arXiv preprint arXiv:2412.14161*, 2024. URL https://arxiv.org/pdf/2412.14161.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W Cohen, Ruslan Salakhutdinov, and Christopher D Manning. Hotpotqa: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 2369–2380, 2018. URL https://arxiv.org/pdf/1809.09600.

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 *The eleventh international conference on learning representations*, 2022.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains, 2024. *URL https://arxiv. org/abs/2406.12045*, 2024. URL https://arxiv.org/pdf/2406.12045.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. Webarena: A realistic web environment for building autonomous agents. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=oKn9c6ytLx.

APPENDIX

A RELATED WORK

Tool-Use and Function-Calling Benchmarks. Early tool-use benchmarks evaluate whether models can select appropriate tools and generate well-formed calls. Gorilla Patil et al. (2024) connects LLMs to large API collections, ToolLLM Qin et al. (2024) scales instruction-following data to thousands of real-world APIs, and the Berkeley Function Calling Leaderboard (BFCL) Patil et al. (2025) standardizes function-calling evaluation across single-call, parallel-call, multi-turn, and multi-step settings. These benchmarks are valuable for measuring API grounding and call-structure correctness, but they primarily view tool use as producing correct invocations. In contrast, `E-Bench` evaluates closed-loop interaction with a persistent environment, where agents must acquire information from observations, reason across steps, and execute actions whose effects modify backend product state.

Stateful Agent Environments for Tool Use Benchmarks. A second line of work evaluates agents in richer, stateful environments, but often compromises controllability, reusability, or automated construction. Benchmarks built on real or live MCP servers, such as MCP-Atlas Bandi et al. (2026), MCP-Universe Luo et al. (2025), MCP-Bench Wang et al. (2026), and LiveMCPBench Mo et al. (2025), provide high ecological validity but are difficult to reset, scale, and use for repeated state-changing evaluation. Controlled simulators avoid live services, yet often couple environments to individual tasks, as in AppWorld Trivedi et al. (2024) and VitaBench He et al. (2025). More broadly, benchmarks such as VitaBench He et al. (2025), τ-bench Yao et al. (2024), ToolSandbox Lu et al. (2025), and MCPMark Wu et al. (2026) require substantial human or engineering effort to package user requests, annotate tasks, create ground truth, or build custom checkers respectively. MCPEval Liu et al. (2025) moves toward automation by generating tasks from MCP tool specifications and verifying them through frontier-agent execution; however, it evaluates agents by alignment to verifier-generated tool-use trajectories, which can favor a particular solution path over the final outcome.

`E-Bench` addresses these limitations by decoupling automatic environment and task construction: each domain is a reusable, controllable, database-backed product world shared across many closed-loop tasks. LLM agents then synthesize tasks by inspecting this environment, executing intended state changes, and recording verified database diffs as trajectory-agnostic ground truth, enabling deterministic grading without task-specific engineering or LLM-based evaluation.

B MORE BENCHMARK DETAILS

B.1 BENCHMARK STATISTICS

Table 3 lists the six capability types annotated on `E-Bench` tasks together with their per-domain task counts.

Table 3: Capability types annotated on `E-Bench` tasks. Each task may compose multiple capabilities.

Capability	What the task requires	HoK	Music	Meeting	Total
Full-Data Acquisition	Retrieve complete hidden state before acting, avoiding decisions from partial observations.	86	92	19	197
Multi-Condition Filtering	Identify target entities satisfying multiple simultaneous constraints or states.	36	20	16	72
Aggregation and Computation	Compute counts, rankings, summaries, or other aggregates over retrieved state.	56	66	37	159
Cross-Step Dependency	Coordinate sequential tool calls where later steps depend on earlier results.	78	65	64	207
Precise Boundary Judgment	Make exact threshold, boundary, ranking, capacity, or availability decisions.	35	15	94	144
Cross-Entity Cascade	Propagate an operation or decision across related entity types.	5	8	93	106

Table 4 reports the per-domain statistics of `E-Bench`, covering the synthetic product environments, the ground-truth database changes, and the solver tools available under each setting.

Table 4: Per-domain statistics of E-Bench. "Principal Entities" stands for "users" for *Honor of Kings* and *QQ Music*, and "employees" for *Tencent Meeting*; "DB Diff" stands for the ground-truth database changes; and "Solver Tools" stands for MCP tools available to the *solver*, where E-Bench-Code additionally grants the `exec_code` tool (one more MCP tool per domain). Note that there is not DB Diff of update type for QQ Music by environment desgin.

	Honor of Kings	**QQ Music**	**Tencent Meeting**	**Total**
# Tables	16	12	13	41
# Columns	168	72	105	345
# Columns / Table [min∼max (mean)]	4∼22 (10.5)	3∼11 (6.0)	2∼22 (8.1)	-
# Foreign-key Relations	29	16	24	69
# Rows	18,646	28,321	29,350	76,317
# Principal Entities	170	54	995	1,219
# Tasks	110	104	109	323
# DB Diffs / Task [min∼max (mean, median)]	3∼25 (9.5, 9)	5∼80 (16.5, 13)	7∼221 (48.5, 35)	-
DB Diff Type [insert / update /delete] (%)	37.7 / 46.1 / 16.2	98.8 / 0.0 / 1.2	98.6 / 0.3 / 1.1	-
# Solver Tools [E-Bench / E-Bench-Code]	33 / 34	27 / 28	25 / 26	-

Table 4 summarizes the scale and diversity of the constructed environments and tasks. Each domain forms a coherent product world rather than a thin collection of tool stubs: the synthetic databases contain 12 to 16 tables connected by 16 to 29 foreign-key relations, with 72 to 168 columns in total and 2 to 22 columns per table (6.0 to 10.5 on average). Populating these schemas yields 18.6K to 29.4K rows per domain (76.3K total) and over 0.6M data cells, organized around 170 users (*Honor of Kings*), 54 users (*QQ Music*), and 995 employees (*Tencent Meeting*) together with the heroes, songs, meetings, and other entities that reference them. Because tasks are synthesized from this shared, pre-populated state rather than task-local fixtures, each environment supports roughly one hundred distinct tasks.

The design goals of each domain are listed below: (1) *Honor of Kings*: A MOBA game platform covering player social interactions, team management, room-based matchmaking, and match review. Agents can query the current user's players, heroes, ranks, friends, teams, rooms, and match records, and perform actions such as adding friends, managing blacklists, creating or dissolving rooms, sending or responding to room invitations, approving team applications, favoriting matches, and purchasing heroes. (2) *QQ Music*: A music platform covering content search, user preferences, and playlist management. Agents can search and inspect songs, artists, albums, playlists, comments, and listening histories, and perform actions such as creating or deleting playlists, adding or removing songs, favoriting songs, and following artists for the current user. (3) *Tencent Meeting*: An enterprise collaboration platform covering organizational structure, group chats, meetings, meeting-room bookings, and employee schedules. Agents can query employees, departments, group chats, meetings, meeting rooms, and schedules, and perform actions such as creating meetings, booking meeting rooms, creating schedules, canceling meetings, managing group-chat members, and transferring employees across departments for the current employee.

The resulting tasks require substantive state changes rather than single-record edits. Each task is bound with a ground-truth database diff for grading, with an average of 24.9 row-level changes across the benchmark, ranging from 9.5 in *Honor of Kings* to 48.5 in *Tencent Meeting*. Individual tasks induce between 3 and 221 changes. Change patterns are domain-dependent: *Honor of Kings* tasks mix inserts, updates, and deletes, while *QQ Music* and *Tencent Meeting* are dominated by inserts, reflecting bulk operations such as adding favorites, booking meetings, and inviting participants. This diversity in scale and edit type requires agents to track many interdependent state changes rather than perform a single localized action. Moreover, as shown in Table 3, different domains emphasize different task categories, leading to differences in both difficulty and evaluation focus.

B.2 Task Construction Details

This appendix expands on the scalable task-construction pipeline (Section 2.3) and the E-Bench-Code setting (Section 2.4).

Intent Rewriting for the Information Gap. The information gap is enforced by an intent-rewriting step that never leaks the generator's privileged view of global database state. Values that are genuinely user-specified—such as a playlist name, a meeting title, or a date range—are kept explicit in the query, whereas values derived from the database are replaced by the *rule* that produces them. For example, the final query asks the agent to "add the unavailable songs in my favorites" rather than listing the corresponding song identifiers or even revealing their count. As a result, the solver must interact with the public tools to recover the same hidden target set.

Tool Gap and Programmatic Generation. During generation, the task generator may call `query_sql`, a read-only SQL tool over the entire database, and `exec_code`, which exposes the public domain tools as callable functions. Within `exec_code`, the generator can write programs that paginate through results, aggregate tool outputs, perform set operations, traverse multi-hop relations, and execute dependent write operations, providing a reliable mechanism for authoring tasks whose ground truth requires computation over the environment rather than a single direct API call. At benchmark execution time, these internal tools are filtered out, leaving only the business-facing APIs, so an evaluated agent must reproduce the same result through ordinary tool calls. In the `E-Bench`-Code setting, this tool gap is partially closed by granting the solver code execution over the public tools, and comparing the base and code settings quantifies how much structured computation raises the agent's effective ceiling. The two settings isolate complementary abilities: `E-Bench` tests manual coordination over extremely long multi-step tool-call sequences, whereas `E-Bench`-Code tests whether agents extract the relevant information.

Generation Loop. Each task is produced through a product-oriented loop—*inspect data* $\rightarrow$ *decide targets* $\rightarrow$ *modify state* $\rightarrow$ *summarize intent*—run for a sampled user and one or more sampled capability types. The generator first explores a temporary database copy, then performs the intended state changes on that copy, and finally emits a structured summary of its selection rules, non-default parameters, free-text values, and write operations. The framework snapshots the temporary database before and after generation and computes the exact ground-truth diff, discarding candidates that produce no meaningful change, fail to yield a parseable intent, or exceed a predefined tool-call cost budget.

Validation Chain. Surviving candidates pass a multi-stage validation chain that retains only executable, non-trivial, and deterministically checkable tasks. First, each candidate must contain a well-formed ground-truth diff. Second, three strong validators—GPT-5.5, Claude Opus 4.7, and GLM-5.1—attempt to solve the task with access to both `query_sql` and `exec_code`, matching the generator's full-information setting. This removes the generation–validation capability gap, so that validation tests the correctness of the recorded state change rather than a validator's ability to recover hidden information under solver-like constraints. A task is accepted only if at least two of the three validators produce database changes consistent with the recorded answer. Third, tasks that a weak baseline solves too easily are filtered out to preserve discrimination among stronger agents. The remaining tasks are stratified by capability type to maintain a balanced distribution of reasoning patterns.

Task Specification. Each accepted task is stored as a self-contained JSON specification containing the user-facing query, the expected database changes (a `db_snapshot_diff` with `insert`, `update`, and `delete` entries), and the capability types the task exercises. Because correctness is checked against deterministic database-state changes rather than an LLM judge, evaluation is stable and avoids the variance of semantic judging.

C More Benchmark Results Analysis

C.1 Parallel Tool Calling and Token Consumption

Parallel tool calling is an important capability for agents operating in stateful environments, and it varies substantially across models. Many tasks in `E-Bench` require agents to explore multiple independent parts of the environment, such as retrieving details for several songs or checking the daily schedules of different users. In these cases, issuing independent tool calls in parallel can substantially improve efficiency by shortening interaction trajectories and reducing repeated context accumulation, thereby lowering token consumption.

We further analyze whether evaluated models can issue multiple tool calls within the same interaction round. For each model, we compute the average number of tool calls per turn as a proxy for parallel tool-calling capability, and compare it with the average token consumption per task. Our token-consumption statistic sums the input tokens sent to the model and the output tokens generated by the model across all model invocations in a trajectory.

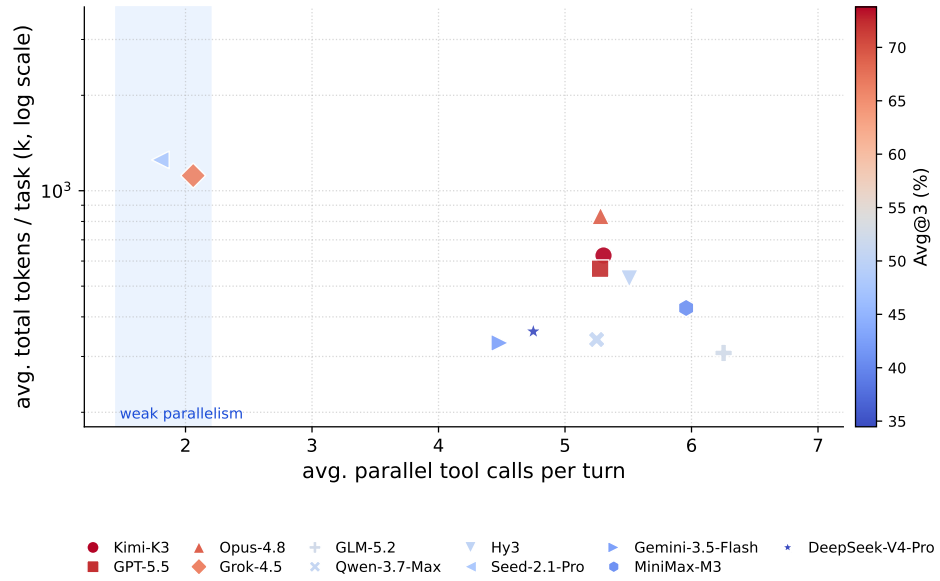


Figure 7: Relationship between global parallel tool-calling capability and token consumption. Each point denotes one evaluated model; marker shape identifies the model, and color indicates Avg@3. The x-axis measures the average number of tool calls issued per turn, and the y-axis shows the average total tokens consumed per task.

As shown in Figure 7, models that issue more MCP tool calls in each turn, for example GLM-5.2, usually finish the task with fewer interaction rounds, thereby avoiding repeatedly re-reading long observations and accumulating excessive context tokens. In contrast, Grok-4.5 and Seed-2.1-Pro exhibit notably weak parallelism. Grok-4.5 issues only 2.06 tool calls per turn on average and consumes 1.12M tokens per task, while Seed-2.1-Pro issues only 1.80 tool calls per turn and consumes 1.25M tokens per task. These two models consume 1.18M tokens per task on average, substantially higher than the 0.44M average of models with at least four tool calls per turn. This suggests that insufficient parallel tool invocation is an important source of token inefficiency in multi-step tool-use agents: weaker parallel tool-calling capability directly increases token usage and therefore leads to higher monetary cost under real API pricing.

C.2 Per-Model Effect of Code Execution

Section 3.5 analyzed how `exec_code` affects performance and interaction cost for the three models with a paired `E-Bench` run shown in the main text (GLM-5.2, Qwen-3.7-Max, Hy3). Here we report the same metrics on different kinds of task for the remaining eight models, in Figure 8.

The pattern in Figure 8 mirrors that of Figure 6 in the main text.

First, tasks for which a model always uses code (all-C) are the most tool-intensive. Under `E-Bench`, where `exec_code` is unavailable, models must issue many individual domain-specific MCP calls to solve these tasks, leading to substantially higher call counts compared to the other two types of tasks. Granting `exec_code` sharply reduces the number of MCP calls; Avg@3 often improves as well, and the number of turns often decreases. This confirms that a single code block can replace a long sequence of individual tool calls without sacrificing accuracy.

A caveat is that some models rely on `exec_code` so heavily that their *partial* and *no-C* groups contain very few tasks: Gemini-3.5-Flash has no tasks outside all-C; GPT-5.5 has no no-C tasks;

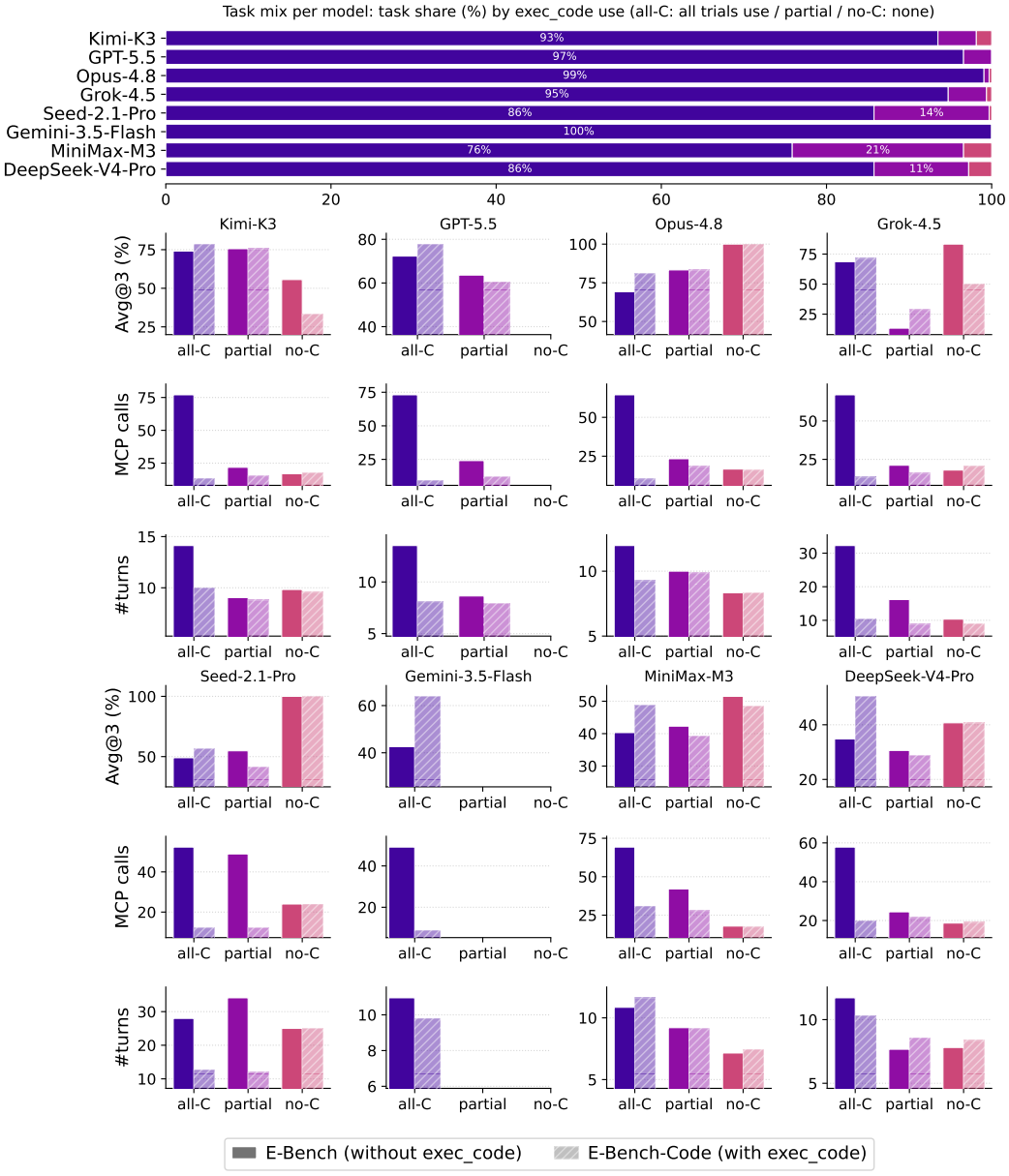


Figure 8: Effect of `exec_code` for the eight models not shown in Figure 6. Bars are absent where a model has no tasks in a group (e.g. Gemini-3.5-Flash used `exec_code` on every task, so it has only the all-C group).

Opus-4.8 assigns 99% of tasks to all-C; Grok-4.5 assigns 95% of tasks to all-C with only two no-C tasks out of 323 tasks in total; and Kimi-K3 likewise assigns 94% of tasks to all-C with only six no-C tasks. Thus, their no-C and partial bars should be interpreted as indicative rather than statistically robust. The trends are clearest for models with sizeable groups in all three categories (Seed-2.1-Pro, MiniMax-M3, and DeepSeek-V4-Pro): for these models, no-C tasks show nearly identical cost and performance across the two settings, as expected when code is not used.

C.3 Does Spending More Help in E−Bench-Code?

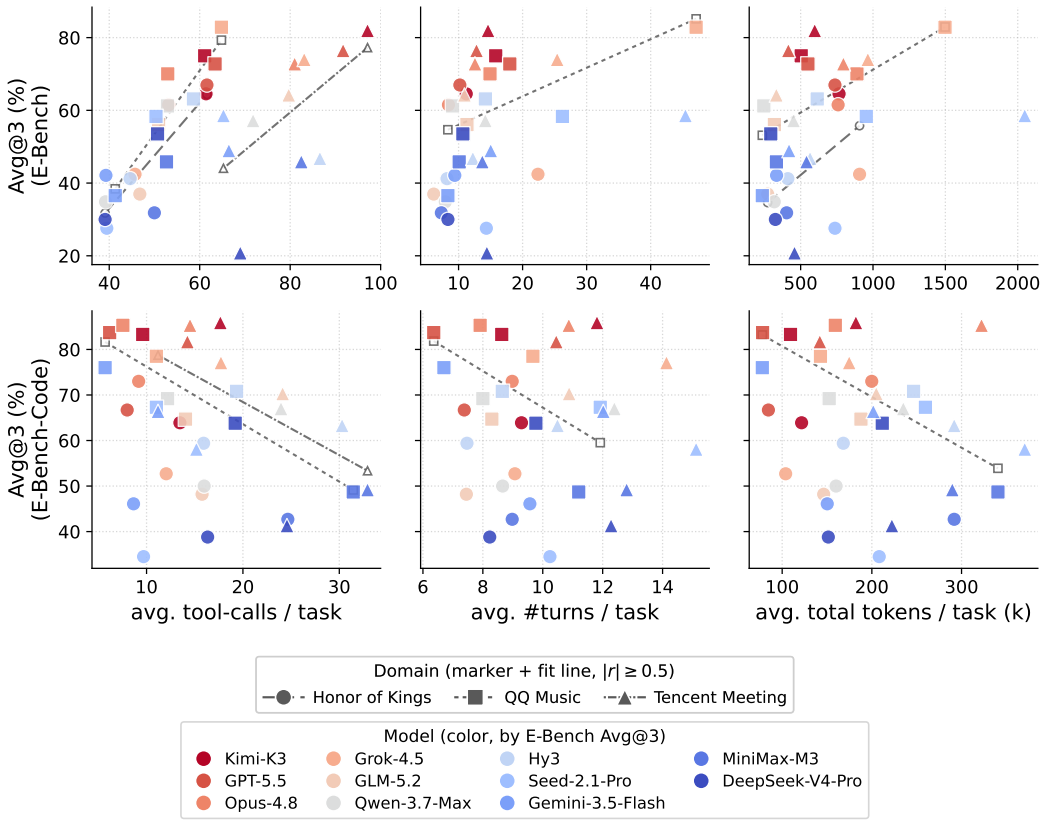


Figure 9: Avg@3 (%) versus per-task interaction cost (task-average MCP tool calls, number of turns, and total tokens) for all evaluated models in three domains under E−Bench (top row) and E−Bench-Code (bottom row). Gray least-squares lines, matching the domain markers in the legend are drawn in any panel where the within-domain Pearson correlation satisfies $|r| \geq 0.5$.

As discussed in Section C.3, under E−Bench, higher interaction cost is associated with better performance (Avg@3) on harder tasks: stronger runs tend to use more MCP tool calls, more agent turns, and more tokens (see also the top row of Figure 9).

Under E−Bench-Code, however, this trend sometimes seems to reverse. With exec_code available, models can fold multiple domain-specific function calls into a single MCP call within one turn. Still, the correlations between Avg@3 and cost are weak: Avg@3 is only weakly negatively correlated with MCP tool calls ($r = -0.36$) and is nearly uncorrelated with turns and total tokens ($r = -0.07$ and $r = -0.25$, respectively). Thus, we cannot conclude that packing more domain-specific calls into exec_code is the only path to success, although it may increase the likelihood of solving demanding tasks efficiently.

Table 5: Per-task cost efficiency of the 11 evaluated models under E−Bench and E−Bench-Code. "p_{in}", "p_{cache}", and "p_{out}" are the fresh-input, cache-hit-input, and output token prices in USD per million tokens. Prices for Hy3, Seed-2.1-Pro, and DeepSeek-V4-Pro are shown to four decimals because these models are priced in RMB and converted to USD manually at the current exchange rate (1:6.77); the remaining models are priced natively in USD and shown to two decimals. Rows follow the canonical E−Bench Avg@3 order.

	Price ($/M)			E−Bench		E−Bench-Code	
Model	p_{in}	p_{cache}	p_{out}	Avg@3	Cost ($)	Avg@3	Cost ($)
Kimi-K3	3.00	0.30	15.00	73.8%	0.634	77.6%	0.191
GPT-5.5	5.00	0.50	30.00	72.0%	1.786	77.2%	0.481
Opus-4.8	6.25	0.50	25.00	68.8%	3.398	81.1%	0.833
Grok-4.5	2.00	0.30	6.00	66.1%	0.504	69.2%	0.137
GLM-5.2	1.40	0.26	4.40	52.3%	0.258	61.0%	0.145
Qwen-3.7-Max	1.25	0.25	3.75	50.9%	0.243	61.9%	0.138
Hy3	0.1477	0.0369	0.5908	50.2%	0.053	64.4%	0.029
Seed-2.1-Pro	0.8863	0.1773	4.4313	47.9%	0.452	53.0%	0.138
Gemini-3.5-Flash	1.50	0.15	9.00	42.6%	0.452	59.1%	0.189
MiniMax-M3	0.30	0.06	1.20	41.1%	0.063	46.9%	0.041
DeepSeek-V4-Pro	0.4350	0.0036	0.8700	34.5%	0.028	47.7%	0.024

C.4 API Cost-Efficiency Analysis

Real deployments care not only about accuracy but also about the monetary cost of reaching it. We therefore report the task averaged API cost alongside Avg@3, under both E−Bench and E−Bench-Code in Figure 1 and Table 5. The task averaged API cost is calculated following: Real-world deployments must balance accuracy against the monetary cost required to achieve it. We therefore report task-averaged API cost alongside Avg@3 for both E−Bench and E−Bench-Code, as shown in Figure 1 and Table 5. The task-averaged API cost is computed as: follows:

$$\text{cost} = \frac{n_{\text{in}} \cdot p_{\text{in}} + n_{\text{cache}} \cdot p_{\text{cache}} + n_{\text{out}} \cdot p_{\text{out}}}{10^6}, \tag{1}$$

where n_{in}, n_{cache}, and n_{out} are the task-average fresh-input (also, cache-activate-input), cached-input, and output tokens taken from our result files, and $p_{\text{in}}, p_{\text{cache}}, p_{\text{out}}$ are the corresponding prices in USD per million tokens (listed per model in Table 5). where n_{in}, n_{cache}, and n_{out} denote the number of task-averaged fresh-input (also, cache-activation input), cached-input, and output tokens, respectively; and $p_{\text{in}}, p_{\text{cache}}, p_{\text{out}}$ denote the corresponding prices in USD ($) per million tokens for each model, as listed in Table 5.

As shown in Table 5 and Figure 1, API cost varies by more than an order of magnitude across models with comparable accuracy. Moreover, exec_code shifts the overall frontier toward the cost-efficient upper-left region: by consolidating many domain-tool calls into compact programs, it substantially reduces per-task token usage and hence cost, while also improving Avg@3.

Under both settings, DeepSeek-V4-Pro, Hy3, Grok-4.5, and Kimi-K3 lie on the Pareto frontier of cost efficiency, achieving relatively high Avg@3 at a given API cost. From this perspective, these models offer the strongest cost-performance trade-offs among the evaluated models.

C.5 Partial Results with Muse-Spark-1.1

We also tested Muse-Spark-1.1 on E−Bench and E−Bench-Code. However, due to its strict content-moderation mechanism, some tasks failed to be tested. For example, in *Honor of Kings*, a task with narration like "delete all that kind of friends, do not leave any of them" consistently triggers a content-policy-violation and cannot be evaluated. Only 265 of the 323 tasks completed successfully across all three trials of *both* settings (268 considering E−Bench alone, 287 considering E−Bench-Code alone). We therefore compare Muse-Spark-1.1 against all other models on these 265 succeeded tasks, re-evaluating every model on the same subset. Results are reported in Table 6.

Table 6: Performance (Avg@3, Pass@3, Pass^3) and interaction cost (number of MCP tool calls and agent action turns) of the 11 evaluated models and Muse-Spark-1.1 on the 265 tasks that Muse-Spark-1.1 completes without content-policy-violation failure in any trial, under `E-Bench` and `E-Bench`-Code. MCP and turns are per-task averages; tokens are per-task input and output tokens in total. Rows follow the canonical `E-Bench` Avg@3 order. In each column, the best three values are highlighted in **bold**, underline, and gray background (highest for performance, lowest for cost).

	`E-Bench`					`E-Bench`-Code				
Model	Avg@3	Pass@3	Pass^3	MCP	#turns	Avg@3	Pass@3	Pass^3	MCP	#turns
Kimi-K3	**71.5%**	**85.7%**	**56.6%**	59.98	12.93	75.8%	85.7%	63.8%	13.32	9.69
GPT-5.5	70.8%	81.1%	56.6%	60.39	12.53	75.3%	85.3%	64.2%	9.31	**7.94**
Opus-4.8	69.3%	84.9%	52.5%	54.48	11.26	**79.9%**	**89.8%**	**66.0%**	10.01	9.03
Grok-4.5	64.7%	79.6%	51.7%	59.04	30.41	66.7%	80.4%	53.6%	13.23	9.90
Muse-Spark-1.1	61.6%	75.8%	46.0%	71.59	15.53	69.9%	84.2%	53.2%	8.79	8.17
GLM-5.2	52.7%	71.3%	32.1%	51.66	**9.02**	59.4%	77.0%	39.2%	17.02	8.45
Hy3	49.6%	74.0%	26.4%	56.02	11.05	62.4%	83.4%	38.9%	20.53	8.58
Seed-2.1-Pro	48.6%	64.5%	33.2%	49.38	25.68	50.7%	70.2%	27.9%	11.64	12.11
Qwen-3.7-Max	48.3%	70.9%	23.4%	50.71	9.72	60.5%	77.7%	41.5%	16.51	9.32
Gemini-3.5-Flash	44.0%	64.9%	24.2%	**45.41**	10.32	59.1%	78.9%	36.6%	**8.27**	9.19
MiniMax-M3	40.6%	61.9%	20.8%	52.94	9.50	45.5%	70.9%	21.1%	27.87	10.26
DeepSeek-V4-Pro	36.5%	56.2%	19.2%	48.80	10.36	47.5%	69.8%	26.0%	19.53	9.64

As shown in Table 6, Muse-Spark-1.1 is a competitive mid-frontier model that benefits clearly from code execution. On `E-Bench`, it achieves 61.6% Avg@3, ranking 5th among the 12 models and clearly outperforming the rest of the field, none of which exceeds 53%. With `exec_code`, its Avg@3 rises to 69.9%, moving it to 4th. As with other models, reliability remains a challenge: Pass^3 is substantially lower than Pass@3. Notably, when `exec_code` is available, Muse-Spark-1.1 shifts from being the most tool-intensive model—issuing 71.59 MCP calls per task, the highest among all 12 models, with heavy token consumption—to one of the most efficient, achieving the second-lowest number of MCP calls and agent turns per task.

Muse-Spark-1.1 is also a strong parallel tool call user. Under `E-Bench`, Muse-Spark-1.1 issues about 4.6 tool calls per turn (71.59 calls over 15.53 turns), placing it among the strong-parallelism models alongside the frontier tier and well above the weak-parallelism outliers (Grok-4.5 and Seed-2.1-Pro, both below 2; see Section C.1). Its high base call count thus reflects genuine parallel invocation within turns rather than long serial trajectories.

D Trajectory Case Studies

We present one Hy3 trajectory case from each of the three E-Bench domains to provide a more intuitive view of our evaluation tasks and the behaviors that the benchmark captures. Each case first shows the original Chinese user query and its English translation. Due to space limitations and for readability, the trajectory visualizations only show the names of tool calls in each round, omitting the system prompt, assistant reasoning content, and assistant content.

D.1 Honor of Kings: Directional State Updates

Case 1: Honor of Kings

Chinese query. 我是辩护人出击，我的ID是WX00000047。帮我整顿一下好友列表，分两件事做：
先找出最近30天里跟我一起打过对局、共同胜率不低于五成、但还没跟我建立亲密关系的好友，从里面挑亲密度最高的那个，把他设成星标好友，同时把亲密关系设成「死党」。
然后，把那些最近30天一场都没跟我打过、亲密度不到200、也没有亲密关系的好友，全部删掉。

English query. I'm "Defense Advocate Strikes", my ID is WX00000047. Help me tidy up my friend list, in two parts:

> First, find the friends who have played matches with me in the last 30 days, whose joint win rate is at least 50%, but who have not yet established an intimate relationship with me. Among them, pick the one with the highest intimacy, set them as a starred friend, and set the intimate relationship to "best buddy".
> Then, delete all the friends who haven't played a single match with me in the last 30 days, whose intimacy is below 200, and who have no intimate relationship with me.

This failed trajectory contains 11 model calls, 10 tool-use rounds, and 46 tool calls, with a maximum of 12 parallel tool calls in one round. Hy3 satisfies 7 out of 8 checks, but fails because of one extra state update. The model observes the field `a_star_b` in the friend-relation tool schema and assumes that it means "I star the other user." However, in this relation record the current user is on the `user_b` side, so the correct direction is `b_star_a=1`. The model later re-checks the relation and writes the correct `b_star_a=1`, but does not clear the previously written `a_star_b=1`. Because E-Bench uses field-level exact state diff for verification, this unexpected extra write is sufficient to make the task fail. This case illustrates why our benchmark stresses not only finding the right entities, but also performing precise state-changing operations. Figure 10 shows the corresponding trajectory.

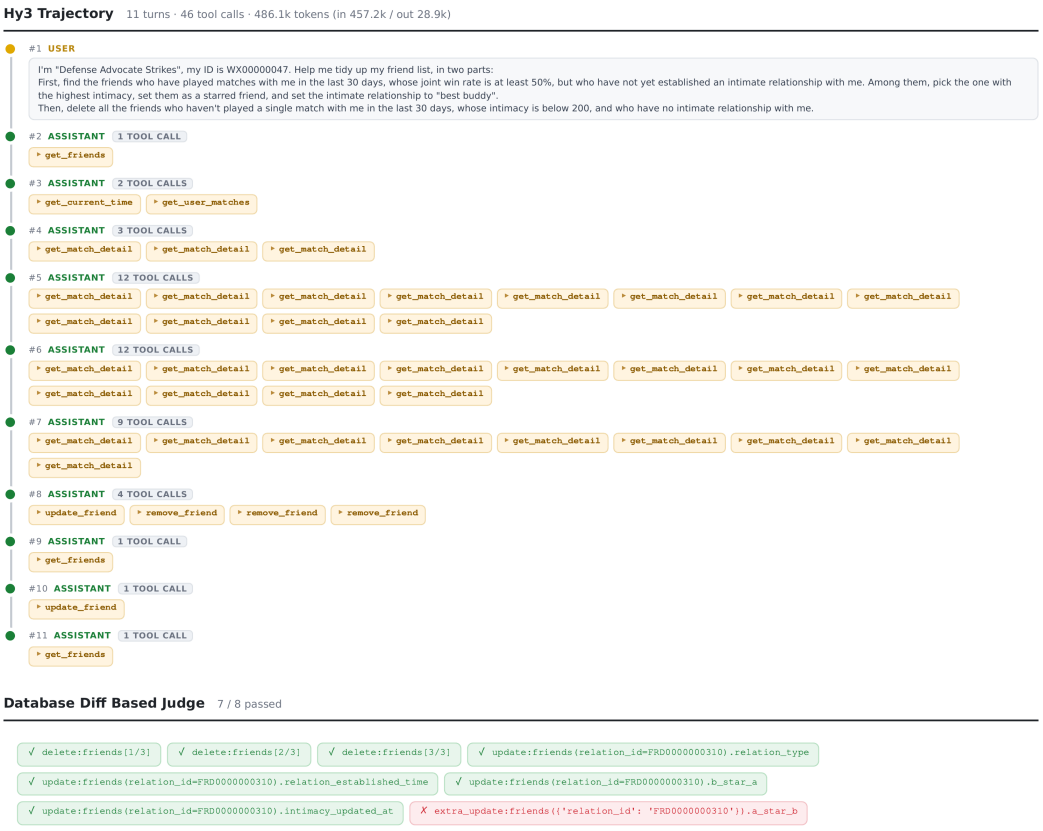


Figure 10: Hy3 trajectory on the Honor of Kings case. The task is almost completed, but one directional friend-relation field is written incorrectly and remains as an extra state change.

D.2 Tencent Meeting: Long-Chain Multi-Entity Orchestration

Case 2: Tencent Meeting

Chinese query. 我是市场营销部渠道运营组的胡伟杰，在深圳鹏创大厦办公。突发渠道事件，要紧急拉一个跨部门对齐小会，麻烦帮我安排一下。参会人包括我自己，再从我所在的渠道运营组里挑除我之外工龄最长的两位深圳在职正式员工，另外从产品与设计中心下面的视觉设计组挑工龄最长的那位深圳在职正式员工，一共四个人，我来当发起人和主持人。时间请在本周4 月21 日周一到4 月25 日周五之间、每天9 点到18 点之间，按半小时为步进扫一下这四人的日程，挑出大

家共同空闲的最早一个连续1 小时时段。会议室就在我们深圳鹏创大厦里找，要普通会议室，能装下4 个人、容量最小的那间。腾讯会议名字就叫『【紧急】渠道-视觉对齐会』，记得加到日历并发邮件通知。会议室按上面的时间同步预定好，四位参会人也都各自建一条会议类型的日程，地点填选中的那间会议室，开启提醒并关联到这场会议。最后再建一个项目群，群名叫『渠道×视觉紧急对齐项目群』，由我从渠道运营组发起，把这四个人拉进来，群描述写『渠道运营组与视觉设计组的跨部门紧急对齐协作群，用于追踪渠道物料与视觉设计相关的协同问题，成员为双方工龄最长的核心骨干。』就行。

English query. I'm Hu Weijie from the Channel Operations Group of the Marketing Department, working at Pengchuang Building in Shenzhen. There's a sudden channel incident and I need to urgently call a small cross-department alignment meeting; please help me arrange it. Attendees include myself, plus the two most senior Shenzhen-based full-time active employees from my own Channel Operations Group besides me, and also the most senior Shenzhen-based full-time active employee from the Visual Design Group under the Product & Design Center—four people in total, with me as the organizer and host. For the time, scan these four people's schedules between Monday April 21 and Friday April 25 this week, from 9:00 to 18:00 each day, in half-hour steps, and pick the earliest continuous 1-hour slot when everyone is free. Find the meeting room within our Pengchuang Building in Shenzhen; it must be an ordinary meeting room that can hold 4 people, the one with the smallest capacity. Name the Tencent Meeting "[Urgent] Channel-Visual Alignment Meeting", remember to add it to the calendar and send an email notification. Book the meeting room for the same time as above; also create a meeting-type schedule entry for each of the four attendees, with the location set to the chosen meeting room, reminders enabled, and linked to this meeting. Finally, create a project group named "Channel × Visual Urgent Alignment Project Group", initiated by me from the Channel Operations Group, add these four people into it, and set the group description to: "A cross-department urgent alignment collaboration group between the Channel Operations Group and the Visual Design Group, used to track collaboration issues related to channel materials and visual design; members are the most senior core backbone from both sides."

This is a representative tool-call-heavy orchestration task. The agent must identify the requester, traverse the organization hierarchy to select attendees under tenure, city, employment-status, and group constraints, scan four calendars in half-hour increments to find the earliest common one-hour slot, select the smallest eligible meeting room, and then cascade a sequence of writes: create the meeting, reserve the room, create linked calendar events for all attendees, create the project group, and add members. Hy3 completes all 15 checks with 94 tool calls across 13 model calls. The trajectory in Figure 3a in the main text demonstrates stable long-chain orchestration over multiple entity types, where each earlier decision constrains subsequent writes.

D.3 QQ Music: Parallel Tool Invocation and Token Cost

Case 3: QQ Music

Chinese query. 我是user_maqiang，把我听过5遍及以上、但还没收藏的歌，全都一口气加到我的收藏里。

English query. I'm user_maqiang. Add all the songs I've listened to 5 or more times but haven't favorited yet, all at once, to my favorites.

This task is semantically simple: find all songs with at least five plays that have not been favorited, and add them to the user's favorites. However, the API does not provide a batch-favorite operation, so the agent must invoke `favorite_song` once for each selected song. Hy3 first identifies 38 target songs and then issues all 38 favorite calls in one parallel round, reaching a maximum parallelism of 38. It finishes the task in 5 model calls and 4 tool-use rounds, passing all 38 checks with 139.5K total tokens. Figure 11 shows the Hy3 trajectory.

For comparison, the corresponding Grok-4.5 trajectory in Figure 12 also succeeds but has much weaker parallelism: it issues at most 10 tool calls in one round and therefore splits the same favorite operations into several serial batches. This increases the trajectory to 10 model calls and 9 tool-use rounds, consuming 225.5K total tokens, about 1.6× Hy3's cost. The case provides a concrete instance of the pattern discussed in Section C.1: weaker parallel tool invocation leads to more rounds, repeated context re-processing, and higher token expenditure.

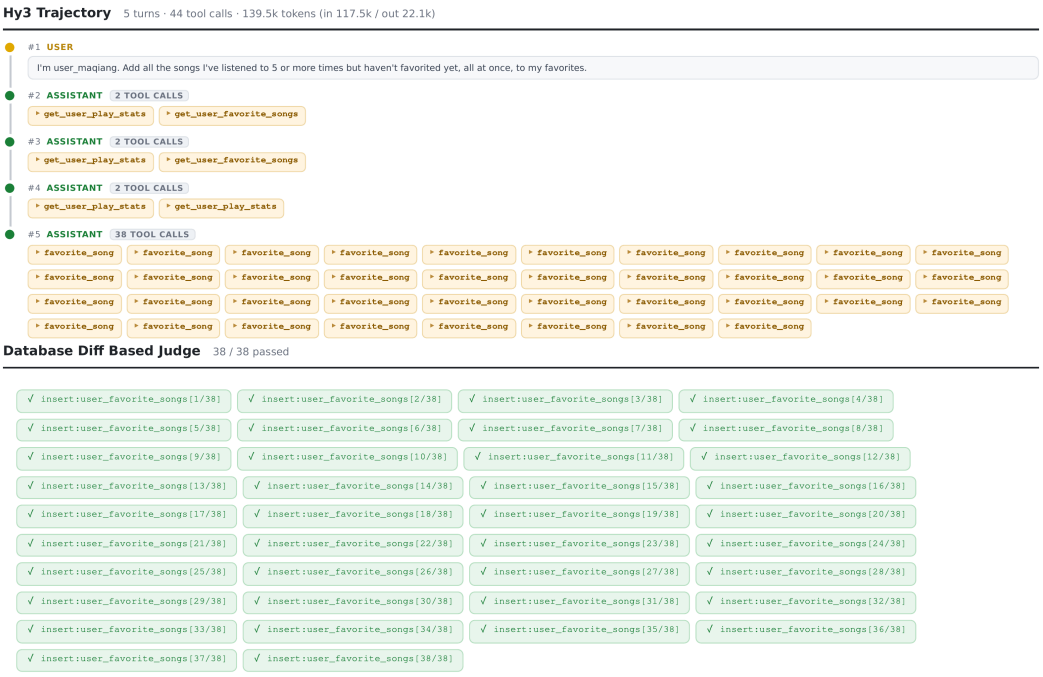


Figure 11: Hy3 trajectory on the QQ Music case. Hy3 completes the 38 favorite operations in one highly parallel round.

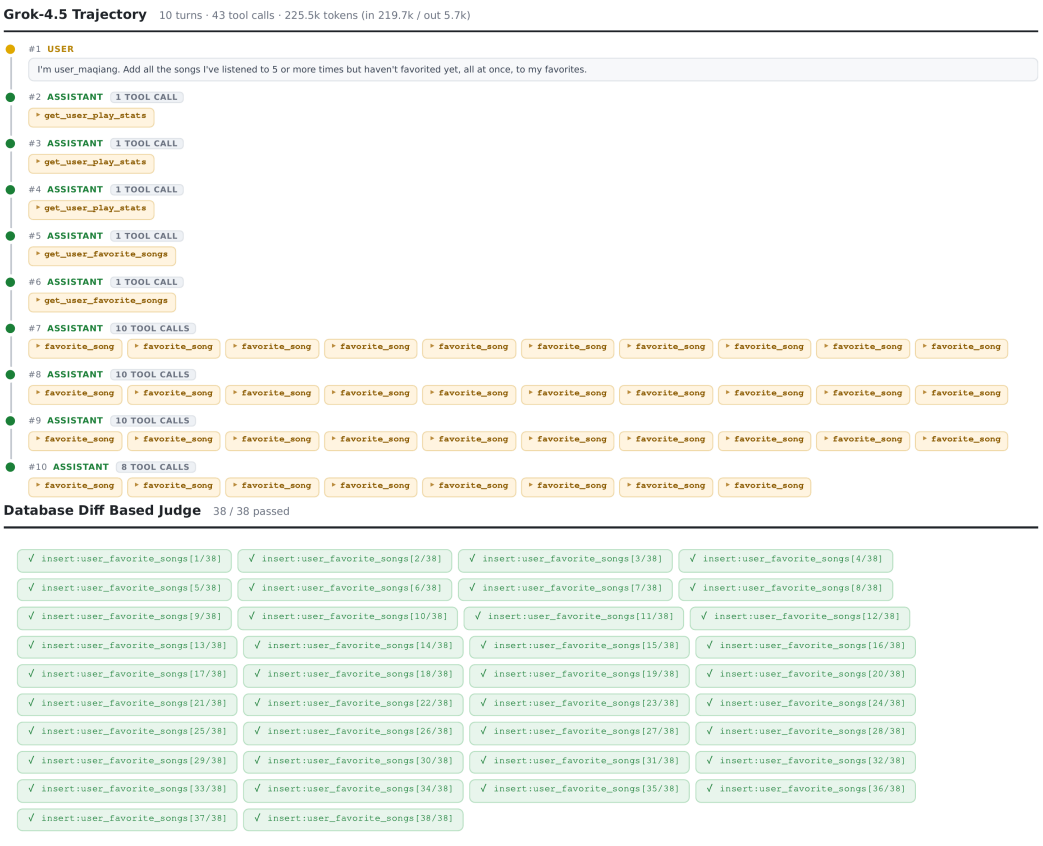


Figure 12: Grok-4.5 trajectory on the QQ Music case. Grok-4.5 executes the same favorite operations in several smaller batches and consumes substantially more tokens.