# THE TOOL DECATHLON: BENCHMARKING LANGUAGE AGENTS FOR DIVERSE, REALISTIC, AND LONG-HORIZON TASK EXECUTION

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Real-world language agents must handle complex, multi-step workflows across diverse applications. For instance, an agent may manage emails by coordinating with calendars and file systems, or monitor a production database like BigQuery to detect anomalies and generate reports following a standard operating manual. However, existing language agent benchmarks often focus on narrow domains or simplified tasks that lack the *diversity*, *realism*, and *long-horizon* complexity required to evaluate agents' real-world performance. To address this gap, we introduce the Tool Decathlon (dubbed as TOOLATHLON), a benchmark for language agents offering diverse applications and tools, realistic environment setup, and reliable execution-based evaluation. TOOLATHLON spans 32 software applications and 604 tools, ranging from everyday platforms such as Google Calendar and Notion to professional applications like WooCommerce, Kubernetes, and BigQuery. Most of the tools are based on a high-quality set of Model Context Protocol (MCP) servers that we may have revised or implemented ourselves. Unlike prior works, which primarily ensure functional realism but offer *limited environment state diversity*, we provide realistic initial environment states from real software. The TOOLATHLON benchmark includes 108 manually sourced or crafted tasks in total, requiring interacting with multiple applications over around 20 turns on average to complete. Each task is strictly verifiable through dedicated evaluation scripts. Comprehensive evaluation of state-of-the-art models highlights their significant shortcomings in performing real-world, long-horizon tasks: the best-performing model, Claude-4-Sonnet, achieves only a 29.9% success rate with 28 tool calling turns on average, while the top open-weights model DeepSeek-V3.1 reaches 13.9%. We expect TOOLATHLON to drive the development of more capable language agents for real-world, long-horizon task execution.

## 1 INTRODUCTION

Tool-based language agents have already demonstrated their impact in real-world domains such as software engineering (Jimenez et al., 2024; The Terminal-Bench Team, 2025), deep research (OpenAI, 2024), and web browsing (Zhou et al., 2024). To further expand the reach of language agents across diverse domains and applications, the Model Context Protocol (MCP) has been proposed to establish a standard for connecting language agents to tens of thousands of applications (Anthropic, 2024).

Existing benchmarks for language agents, however, are restricted to limited domains and tools (Mialon et al., 2023; Liu et al., 2024; Ma et al., 2024; Jimenez et al., 2024; Zhou et al., 2024; Yao et al., 2025; Wei et al., 2025; Xu et al., 2025). By contrast, real-world tasks often require switching across various applications. For example, as demonstrated in Figure 1 (Example #2), a company's administrative agent may need to monitor a real Snowflake database for customer tickets, locate the appropriate PDF operation manual containing instructions on how to identify and handle overdue tickets, and then send the required emails to managers and customers in accordance with the manual. Importantly, this diversity gap extends far beyond differences in tool names or descriptions. The diversity and complexity of environment states across applications, compounded by interaction with them in long trajectories, present substantial challenges for generalization.

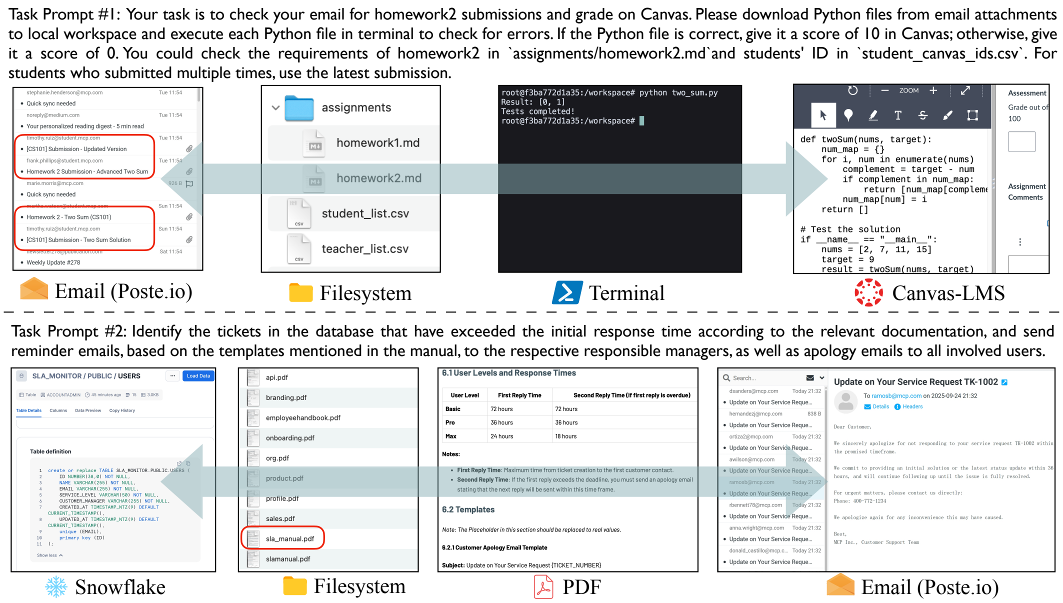



Figure 1: Two examples and the initial environment states in TOOLATHLON. We showcase real-world environment interaction (§2.2) and realistc state initialization (§2.3) here.

To address these challenges, we introduce the Tool Decathlon (TOOLATHLON), a benchmark for evaluating language agents on *diverse, realistic, and long-horizon* tasks. TOOLATHLON spans 32 real-world applications and 604 tools across 108 tasks, covering a wide spectrum of domains ranging from daily affair and education to technology and finance. Tasks are grounded in realistic scenarios and mostly require coordinating multiple applications. Each task is fully verifiable with a dedicated, deterministic evaluation script, comparing outcomes against either static or dynamically generated ground-truth states (e.g., tasks involving the latest NVIDIA shareholder information or real-time train schedules). All tools in TOOLATHLON are sourced from the real world, with the majority obtained from MCP servers.

To faithfully capture the realism and complexity of practical environment states, we tried to adopt the most representative applications such as Google Sheet, Gmail, and Snowflake. However, some remote environment states are difficult to set up to mimic real scenarios. For example, simulating a Canvas course with tens of students would require registering a real account for each student and resetting the states at each evaluation run. Therefore, while we adopt the commonly used applications most of the time, we also incorporate several local pieces of software deployed via containers for convenient and complex environment simulation, such as poste.io for email management to replace Gmail and WooCommerce for online ecommerce platform to replace Shopify. These services are based on their respective open-source software, providing complex observations while allowing us to set up the states within seconds. This stands in stark contrast with simplified or artificial environment states as in prior benchmarks (Patil et al., 2025). In addition, task prompts in TOOLATHLON are crafted to mirror authentic user queries, which are often concise and fuzzy. Models must therefore infer user intent and autonomously devise plans to accomplish tasks, an example is shown in Figure 3.

Concurrent with this work, several MCP-based tool-use benchmarks have emerged (Liu et al., 2025; Mo et al., 2025; Yan et al., 2025; Yin et al., 2025; The MCPMark Team, 2025), but they do not match TOOLATHLON in its reflection of real-world complexity. Some rely on LLM judges without verifiable tasks (Mo et al., 2025; Yin et al., 2025), while others cover few domains or mostly single-application tasks. For instance, MCPUniverse (Luo et al., 2025) spans only six domains, with 90% of tasks involving one app and synthetic initial states, yielding simplified, short interactions (<8 turns). Similarly, MCPMark (The MCPMark Team, 2025) includes only five apps and overly detailed prompts (Figure 3). A full comparison is shown in Table 1.

TOOLATHLON includes a lightweight framework for automated, safe, and scalable evaluation. Each task comes with initial states setup if needed as well as an evaluation script (Figure 2). Executing and evaluating each task is isolated in separate containers to prevent interference. This enables fast parallel

Table 1: Comparison of Tool-Based Language Agent Benchmarks, where BFCLv3-MT represents the multi-turn subset released in the 3rd version of BFCL. "# Apps" denotes the number of MCP servers, which we do not annotate for benchmarks without clear application definition. "Avg # Turns" denotes the number of tool calling turns made by Claude-4-Sonnet, which we use as a proxy for task complexity. "Real Env" (§2.2) means the environment states and observations are from real-world software rather than artificial databases. "States Init" (§2.3) indicates the evaluation begins with a state initialization. "Verifiable Execution" (§2.4) denotes that models need to execute the tools and final results are evaluated based on states. "Realistic Fuzzy Prompt" represents that the task instructions are often fuzzy and ambiguous to mimic real user input (§3.1). *For MCPUniverse, only 10% of the tasks are cross-App. For ACEBench and LiveMCPBench, only $<$10% of the tasks contain simple states initialization. In-depth discussion of these related works is in Appendix B.

| Benchmark | # Tasks | # Apps | Avg # Turns | Real Env | States Init | Verifiable Execution | Cross-App Task | Realistic Fuzzy Prompt |
|---|---|---|---|---|---|---|---|---|
| $\tau$-Bench | 165 | 2 | – | × | ✓ | ✓ | × | × |
| BFCLv3-MT | 800 | – | 3.8 | × | ✓ | ✓ | ✓ | × |
| ACEBench | 2000 | – | 1.7 | × | Partial* | ✓ | × | × |
| AppWorld | 750 | 9 | – | ✓ | ✓ | ✓ | ✓ | × |
| MCPWorld | 201 | 10 | – | ✓ | ✓ | ✓ | × | × |
| MCP-RADAR | 507 | 6 | – | ✓ | ✓ | ✓ | × | × |
| MCPEval | 676 | 19 | – | ✓ | × | × | ✓ | × |
| LiveMCPBench | 95 | 70 | 5.6 | ✓ | Partial* | × | ✓ | × |
| MCP-AgentBench | 600 | 33 | – | ✓ | × | × | ✓ | × |
| LiveMCP-101 | 101 | 41 | 5.4 | ✓ | × | × | ✓ | × |
| MCPAtlas | 1000 | 40+ | 3-6 | ✓ | ✓ | × | ✓ | × |
| MCPUniverse | 231 | 11 | 7.5 | ✓ | × | ✓ | Partial* | × |
| MCPMark | 127 | 5 | 18.5 | ✓ | ✓ | ✓ | × | × |
| **TOOLATHLON** | 108 | 32 | 26.8 | ✓ | ✓ | ✓ | ✓ | ✓ |

evaluation—for example, running Claude-4-Sonnet on all 108 tasks takes only 70 minutes using 10 parallel processes. With extensive experiments on TOOLATHLON, the best-performing models, Claude-4-Sonnet and GPT-5, achieve only 29.9% accuracy, highlighting the unique challenges posed by TOOLATHLON. DeepSeek-V3.1 (DeepSeek-AI, 2025) achieves 13.9% success rate as the best performer among open-source models. Further analysis reveals that weaknesses in long-context modeling and robust tool calling error tracking are major challenges for all evaluated models. We will fully open-source the benchmark and the TOOLATHLON environment, aiming for TOOLATHLON to accelerate the development of practical language agents.

## 2 THE TOOLATHLON ENVIRONMENT AND EVALUATION FRAMEWORK

### 2.1 TASK DEFINITION

Each task in TOOLATHLON can be formulated as a partially observable Markov decision process (POMDP) $(\mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{T}, \mathcal{R}, \mathcal{U})$ with state space $\mathcal{S}$, action space $\mathcal{A}$, observation space $\mathcal{O}$, transition function $\mathcal{T} : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S} \times \mathcal{O}$, reward function $\mathcal{R} : \mathcal{S} \rightarrow [0, 1]$, and instruction space $\mathcal{U}$. The environment states (§2.2, §2.3) can be the status in the email inbox and the observations are the sequential input to the model. The action space $\mathcal{A}$ is the available tools for the respective task and the tool implementation directly defines the transition function. The reward function $\mathcal{R}$ (§2.4) represents our execution-based evaluation which directly evaluates the environment state. Intuitively, real-world tools and environments will yield significantly more complex and diverse states and observations than the synthetic ones, and we will detail our designs of these variables in the following sections.

### 2.2 TOOLS, ENVIRONMENTS, AND FRAMEWORK

**MCP Servers:** In TOOLATHLON, we source our tools through a variety of MCP servers. Specifically, we first decide a list of valuable and common real applications that we aim to benchmark on, then we see if we can find the corresponding open-source MCP servers for them. If not, we implement the MCP servers by ourselves. Notably, many open-source MCP server implementations contain bugs or exhibit certain limitations, for example, without the tools needed to complete our tasks. We further refine and improve these implementations ourselves. This way, we obtain a high-quality set of 32 MCP servers in total, where we include a complete list and their sources in Appendix C. The applications span diverse domains, extending well beyond common daily-use applications such as

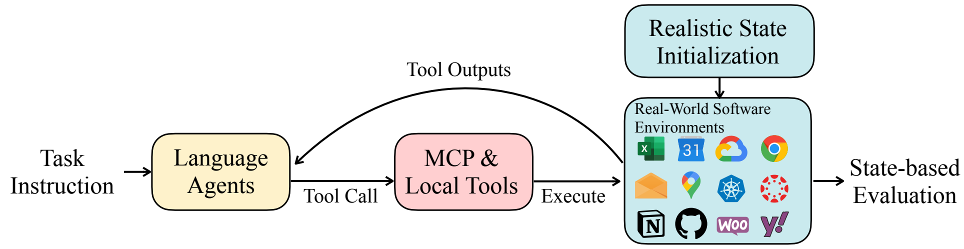



Figure 2: Overview of the TOOLATHLON evaluation framework.

Google Maps, Notion, and Google Calendar, and we also incorporate a number of professional and domain-specific applications to evaluate language agents in high-value productivity scenarios, such as Snowflake for enterprise data management and Kubernetes for cluster management. Although the majority of tools are sourced from MCP servers, the benchmark usage itself is not tied to MCP employment from the model developer side.

**Remote and Locally Containerized Environments:** While tools provide an interface for interacting with environments, they do not directly constitute the environments. Many real-world tools interact directly with existing, remote environments, such as Google Sheets, Google Calendar, Notion, and Gmail. Although remote environments require no implementation effort, they introduce significant challenges when benchmarking tasks that involve modifying environment states. For instance, simulating a realistic Gmail inbox with hundreds of emails from diverse senders would require registering hundreds of Google accounts for every benchmark user, and this inbox would need to be reset prior to each evaluation run. Previous works have attempted to bypass this issue by only supporting read operation to the states (Mialon et al., 2023), or implementing simplified synthetic data structures to mimic environment states (Patil et al., 2025; Yao et al., 2025), but such approaches drastically reduce realism and fail to reflect the complexity of real software environments. In contrast, in TOOLATHLON we leverage both remote environments and locally containerized, open-source applications. Specifically, we deploy the open-source Poste.io for email management, Canvas for course administration, Kubernetes for cluster orchestration, and WooCommerce for e-commerce management. By hosting these realistic applications locally within containers, we can efficiently set up dozens of accounts and initialize complex environment states during evaluation. Compared with existing dedicated agent sandboxes such as SWE-Bench (Jimenez et al., 2024), our environments are more diverse and encompass a wider range of software.

**Agent Framework:** We implement a simple agent framework based on the OpenAI Agents SDK to conduct the agent action loop – at each turn, the model is expected to (optionally) reason and make tool calls. We make several enhancements to improve its basic setup for a more robust workaround to evaluate language agents, including *tool error handling*, *overlong tool response handling* and *context history management*. We also equip this framework with some basic yet common local tools like *python execution*, *web search*, *claim done* and *sleep*. The details can be found in Appendix D.

### 2.3 INITIAL STATE SETUP

In real world, tasks are rarely executed from an empty environment state (e.g., an empty inbox). Instead, agents are typically required to operate based on pre-existing environment states. In agentic scenarios, task difficulty is determined not only by the task instructions but also by the underlying environment states. For example, operating on a folder with only one file to be sued is easier than working with 10 mixed useful and unrelated files (Figure 1, example #2), even if the task descriptions are nearly identical. To capture this, for tasks in TOOLATHLON that starts with an initial state,[1], each of these tasks is equipped with a state initialization script to set up the states at running time, or (and) an `initial_workspace` directory containing pre-set files. Figure 1 and Figure 3 showcase such initial states. When constructing these initial environment states, we design them to closely reflect realistic scenarios. Notably, only very few previous benchmarks have incorporated realistic initial state construction before entering the agent loop, as summarized in Table 1. By contrast, most existing benchmarks start from empty state or overly simplified environment states, thus failing to capture the full complexity of real-world task execution.

[1]As shown in Table 2, 67% of the tasks fall into this category.

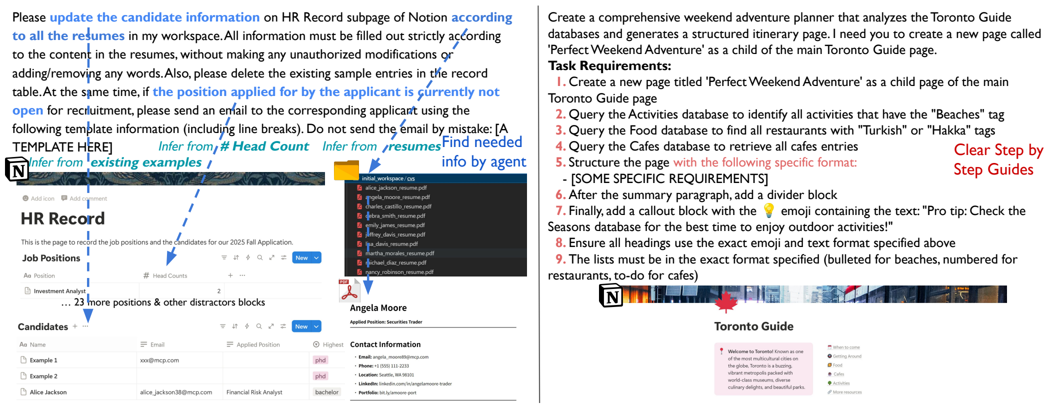



Figure 3: Example task instructions from our benchmark (Left) and MCPMark (The MCPMark Team, 2025) (Right). Ours contain more fuzzy intent that the model need to infer from the environment states.

### 2.4 Reliable Execution-based Evaluation

First, unlike some traditional tool-calling benchmarks that measure single-step tool call accuracy given a fixed context without actual execution (Patil et al., 2024), we think that execution-based evaluation is essential for reliably assessing language agents in realistic scenarios. Second, while many existing benchmarks rely on LLMs as judges to score agent trajectories (Gao et al., 2025; Yin et al., 2025), we contend that verifying the final environment states using deterministic rules offers a far more reliable and reproducible evaluation framework, as demonstrated in several widely adopted agent benchmarks (Zhou et al., 2024; Xie et al., 2024; Jimenez et al., 2024). To achieve this, each task in TOOLATHLON is equipped with a unique, manually crafted evaluation script that ensures precise and consistent measurement of task success. The script may perform robust matching against a static snapshot of the ground-truth environment or follow a reference execution workflow to dynamically retrieve and match real-time information (e.g., NVIDIA shareholders). During evaluation, each task is associated with a configuration file that specifies only the necessary MCP servers ($< 10$) and tools available for use rather than all 32 MCP servers. For each selected MCP server, we do not further filter or retrieve a subset of tools but load all tools within it, which is consistent with the current state of mainstream agent frameworks. Intuitively, providing the model with a larger set of unrelated tools increases task difficulty, as the agent must identify the relevant tools while ignoring distracting ones.

**Safe and Efficient Parallel Evaluation in Containers:** Our TOOLATHLON evaluation framework supports parallel execution to enable efficient model evaluation. Our framework launches each task inside a separate container in parallel, providing strict workspace isolation. On a standard Linux cluster with 16 CPUs and 64 GB of memory, we are able to evaluate Claude-4-Sonnet on 108 tasks in just about 70 minutes of wall time using 10 parallel processes. This demonstrates that TOOLATHLON is both convenient and efficient for practical use by model developers.

## 3 The Toolathlon Tasks

### 3.1 Task Sourcing and Fuzzy Task Instruction

The authors of this work, who are researchers and senior undergraduate students in computer science, source and implement the tasks. We carefully design and adhere to several principles when collecting tasks: (1) **Real User Demands:** All tasks are either directly sourced from real-world websites or crafted to reflect genuine user demands. (2) **Multi-App Orchestration:** We intentionally source tasks that require interaction with multiple MCP servers, as this reflects authentic human workflows and increases task complexity. (3) **Diversity:** To ensure broad task diversity, we adopt a two-stage sourcing process. In the first stage, we start with an initial MCP server list covering more than 50 applications and freely source tasks without restricting to specific servers. In the second stage, we analyze the distribution of the sourced tasks and identify Apps that are important but underrepresented. We then conduct an additional round of targeted task sourcing specifically for them.

**Realistic Fuzzy Task Instruction:** We design task instructions to resemble authentic user input, which is often fuzzy or ambiguous. This requires the agent to infer user intent given the environment state, formulate plans, and carry them out. For example, as shown in Figure 3 Left, a real user may simply say "`Please update the candidate information on HR Record subpage according to all the resumes...... if the position applied for by the applicant is currently not open, ......`" This is a fuzzy instruction without specifying in which format the agent should fill in the information, but the Notion database has provided some examples that the agent needs to know to check itself. Also, the instruction does not mention where to find the status of the posted job and the agent needs to check Notion to find that by itself. In contrast, task instructions in some existing benchmarks (Figure 3 Right) explicitly include detailed step-by-step plans, which reduce the planning effort for agents. More examples of this kind are shown in Figure 10 and 11.

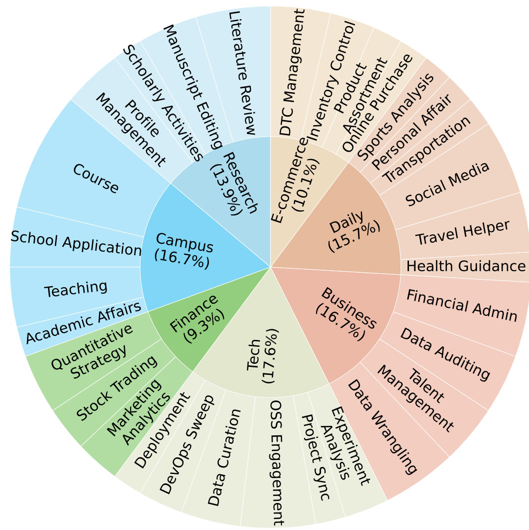


Figure 4: Task topic distribution of TOOLATHLON.

All the sourced tasks experience multiple rounds of rigorous quality check, filtering and refinement before we implement them, and finally we obtain 108 tasks in total. The topic distribution of all tasks is shown in Figure 4 and Table 2 shows some key statistics of the complete benchmark.

### 3.2 TASK IMPLEMENTATION

As described in §2.4, each task in our benchmark is implemented with a corresponding evaluation script and potential initial states setup. This process involves collecting ground-truth states statically or dynamically, and design scripts to automatically clear and re-fill new initial states. To ensure realistic setups and reliable evaluation, implementing a single task in TOOLATHLON requires, on average, 4–6 hours of work by a graduate student majoring in computer science.

Table 2: Key statistics of TOOLATHLON.

| **Statistics** | **Value** |
|---|---|
| # MCP servers (# tools) | 32 (604) |
| # Local toolkits (# tools) | 7 (16) |
| Avg/Min/Max tools per task | 69.9/28/128 |
| Tasks with state initialization | 72/108 (67%) |

**Finalizing Tasks and Quality Check:** After crowd-sourcing task implementations from multiple contributors, we perform intensive quality checks conducted by 5–6 experienced authors. In this stage, each task is carefully reviewed and revised to unify standards across all tasks and ensure correctness, solvability, and unambiguity, which requires approximately 5 hours of labor per task per round of checking. Once all tasks are finalized, we perform an additional round of comprehensive cross-checking and bug fixing of the entire benchmark before running the final experiments.

## 4 EXPERIMENT

In this section, we present the configuration details and experimental settings for several leading commercial and open models on TOOLATHLON, as well as their performance.

### 4.1 SETUP

**Models and Configuration:** Our evaluation includes the leading commercial model series in terms of agentic abilities, such as GPT-5 (-mini) (OpenAI, 2025b), o3&o4-mini (OpenAI, 2025a), Claude-4-Sonnet (Anthropic, 2025), Gemini 2.5(-Pro,-Flash) (Comanici et al., 2025), Grok-4(-Fast) (xAI, 2025a;b), Grok-Code-Fast-1 (xAI, 2025c). We also benchmark the best-performing open-weight

Table 3: Main results for all the models. P@1, P@3, P^3 and # Turns represents Pass@1, Pass@3, Pass^3 and average numbers of turns, respectively. We make **bold** the highest score.

| Model | Research | Campus | Finance | Tech | Business | Daily | E-com | **P@1** | **P@3** | **P^3** | **# Turns** |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | *Proprietary Models* | | | | | | |
| Claude-4-Sonnet | 37.8 | 22.2 | 33.3 | 24.6 | 40.7 | 27.5 | 24.2 | $\mathbf{29.9}_{\pm 3.5}$ | **42.6** | 17.6 | 26.8 |
| GPT-5 | 22.2 | 27.8 | 26.7 | 49.1 | 31.5 | 25.5 | 15.2 | $29.6_{\pm 1.5}$ | 38.9 | **21.3** | 19.5 |
| GPT-5-high | 17.8 | 24.1 | 23.3 | 40.4 | 29.6 | 27.5 | 24.2 | $27.5_{\pm 1.9}$ | 38.0 | 13.9 | 20.8 |
| Grok-4 | 33.3 | 16.7 | 16.7 | 33.3 | 27.8 | 17.6 | 42.4 | $26.5_{\pm 2.3}$ | 39.8 | 14.8 | 19.4 |
| Grok-4-Fast | 26.7 | 14.8 | 13.3 | 22.8 | 22.2 | 17.6 | 27.3 | $20.7_{\pm 0.4}$ | 29.6 | 13.0 | 15.2 |
| o3 | 15.6 | 16.7 | 10.0 | 31.6 | 24.1 | 17.6 | 3.0 | $18.5_{\pm 0.8}$ | 25.9 | 13.0 | 19.2 |
| Grok-Code-Fast-1 | 24.4 | 11.1 | 10.0 | 19.3 | 18.5 | 13.7 | 21.2 | $17.0_{\pm 1.9}$ | 25.9 | 8.3 | 20.4 |
| o4-mini | 15.6 | 9.3 | 16.7 | 19.3 | 14.8 | 11.8 | 6.1 | $13.6_{\pm 0.4}$ | 24.1 | 6.5 | 18.3 |
| GPT-5-mini | 17.8 | 5.6 | 16.7 | 17.5 | 14.8 | 7.8 | 3.0 | $12.0_{\pm 0.8}$ | 19.4 | 5.6 | 19.5 |
| Gemini-2.5-Pro | 6.7 | 3.7 | 3.3 | 14.0 | 7.4 | 2.0 | 0.0 | $5.9_{\pm 1.9}$ | 11.1 | 0.9 | 12.5 |
| Gemini-2.5-Flash | 2.2 | 0.0 | 10.0 | 5.3 | 5.6 | 2.0 | 9.1 | $4.3_{\pm 1.6}$ | 6.5 | 2.8 | 8.8 |
| | | | | | *Open-Source Models* | | | | | | |
| DeepSeek-V3.1 | 2.2 | 13.0 | 23.3 | 21.1 | 14.8 | 11.8 | 12.1 | $13.9_{\pm 0.8}$ | 21.3 | 8.3 | 29.9 |
| GLM-4.5 | 13.3 | 11.1 | 16.7 | 17.5 | 5.6 | 15.7 | 15.2 | $13.3_{\pm 1.9}$ | 20.4 | 8.3 | 23.9 |
| Qwen-3-Coder | 8.9 | 11.1 | 10.0 | 15.8 | 14.8 | 9.8 | 12.1 | $12.0_{\pm 2.0}$ | 19.4 | 7.4 | 29.9 |
| Kimi-K2-0905 | 11.1 | 9.3 | 13.3 | 14.0 | 7.4 | 9.8 | 18.2 | $11.4_{\pm 2.7}$ | 16.7 | 5.6 | 26.7 |

models including Qwen-3-Coder (Qwen Team, 2025), DeepSeek-V3.1 (DeepSeek-AI, 2025), Kimi-K2-0905 (Kimi Team et al., 2025) and GLM-4.5 (GLM-4.5 Team et al., 2025). As described in §2.4, each task is preconfigured with a list of MCP servers and common tools to access. During evaluation, we set the maximum allowable number of turns as 100 for all models. In our main evaluation setup, we only provide the models with the MCP servers and common tools that are useful for executing the task. We note that as each MCP server is equipped with multiple related tools, so the models will still see many unnecessary tools during evaluation. In Appendix E.2, we analyze the performance variation by offering more unrelated MCP servers to the models.

**Metrics:** We evaluate each model three times and report the average pass@1 success rate as well as the standard deviation. We also include the pass@3 – the fraction of tasks with at least one correct trajectory, and pass^3 (Yao et al., 2025) – the fraction of tasks where all three trajectories are correct, to measure the model's potential capability coverage and its ability to complete tasks reliably. We also report the average number of turns used.

### 4.2 Main Results

Results in Table 3 show that Claude-4-Sonnet and GPT-5 are the leading ones, but they only achieve about 30% success rate. Grok-4, with 26.5% Pass@1, is clearly in the second tier. All other models remain at 20% or below. This indicates that our benchmark remains challenging for state-of-the-art models and effectively distinguishes their capabilities. For open-source models, scores remain below 15%, revealing a gap with proprietary models. Notably, grok-code-fast-1 and grok-4-fast completed all tasks at a high speed within 1 hour while scoring near 20% points, whereas others typically required 1-3 hours. Interestingly, increased reasoning effort for thinking models (e.g., GPT-5 vs. GPT-5-high) shows no benefits, suggesting that exploring new observations matters more than extended internal reasoning in agentic tasks. We also find that Gemini-2.5's ability to understand requirements and proactively explore is insufficient – it may neglect requirements or give up prematurely during the execution process, resulting in poor performance when handling complex tasks.

Looking at performance across task categories: Claude-4-Sonnet excels in *Finance*, *Research*, *Business*, and *Daily* tasks, demonstrating strong general capabilities across diverse tools; GPT-5 performs exceptionally in *Campus* adn *Tech* tasks, showcasing effectiveness in educational and technical scenarios; Grok-4 stands out in *E-commerce*, indicating specialized strength in commercial and transactional operations. We also observe significant differences between Pass@3 and Pass^3 success rates. This indicates that while many models have certain capability coverage, they lack consistency in producing reliable results. For real-world tasks, building agents with both high success rates and robust consistency remains a critical challenge.

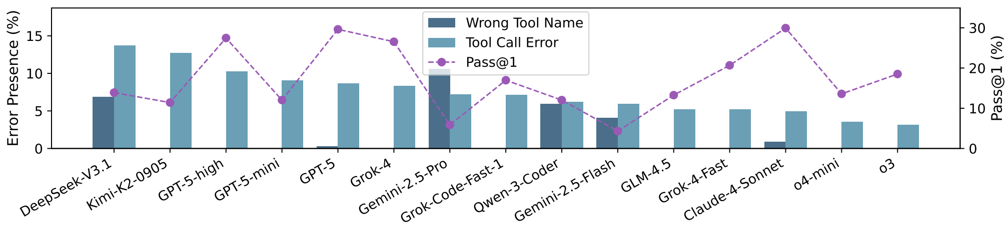



Figure 5: Two kinds of tool calling error presence ratios in calling tools for different models.

## 5 Analysis

In this section, we conduct analysis in depth to better understand model performance in TOOLATHLON, focusing on tool-call errors, as well as long-context and overlong-output challenges. More analysis, including how tool error and the involvement of unrelated MCP servers impact model performance, and qualitative analysis & case studies, can be found in Appendix E and G.

### 5.1 The Failure of Calling Tools

We mainly focus on two major tool-calling errors: hallucinating non-existing tools (e.g., incorrect tool names) and errors raised during tool execution. Statistics for different models on these two types of errors are shown in Figure 5. It can be seen that all models produce tool execution errors to varying degrees, possibly due to incorrect parameter passing or attempts to access non-existent resources. However, we found no significant correlation between overall success rate and the frequency of such errors. In fact, error messages from tools may help models understand the tool implementation or structure, allowing adjustments in subsequent turns. The other type of error–incorrect tool names–more likely affects final scores. Leading models produce few tool name errors. In Appendix E, Figure 9, we further analyze the success rate difference between trajectories containing tool-calling errors versus error-free trajectories. Most models do suffer from tool call errors, except for Claude-4-Sonnet, which may understand the tool better via error messages and gets a higher success rate.

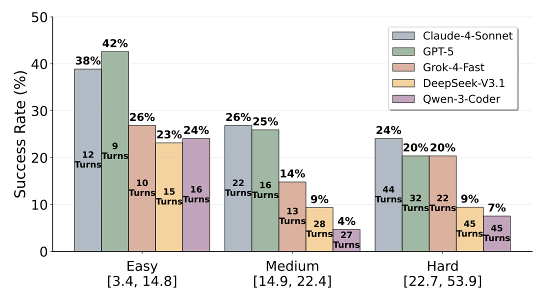



Figure 6: Model performance on three groups of tasks divided by average turns. The x-axis represents different task difficulty groups determined by different avg turns range [Min Turns, Max Turns]

### 5.2 The Long-Context Challenges for Language Agents

Since our benchmark is built on real tools and environments, it naturally generates many long-horizon trajectories. To quantitatively describe differences between tasks, we calculate the average execution turns for each task across all models, and use this as a proxy to divide all tasks into three equally-sized groups: `Easy`, `Medium`, and `Hard`, with execution turns increasing with difficulty. In Figure 6, we show the performance of five representative models on different groups and their average turns in each group. It shows that groups with higher average turns generally have lower success rates across models, and leading models (GPT-5, Claude-4-Sonnet) maintain clear advantages in all groups. We also find that there is no significant difficulty difference between `Medium` and `Hard` groups, suggesting that our benchmark's difficulty doesn't entirely stem from standard multi-step long-horizon execution, but possibly from models ending tasks prematurely without exploring enough observations, leading to failure.

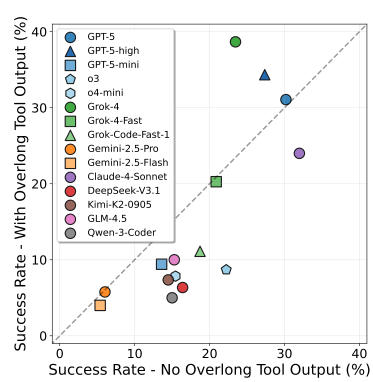



Figure 7: Avg. Success Rate on Trajectories w/wo overlong tool outputs.

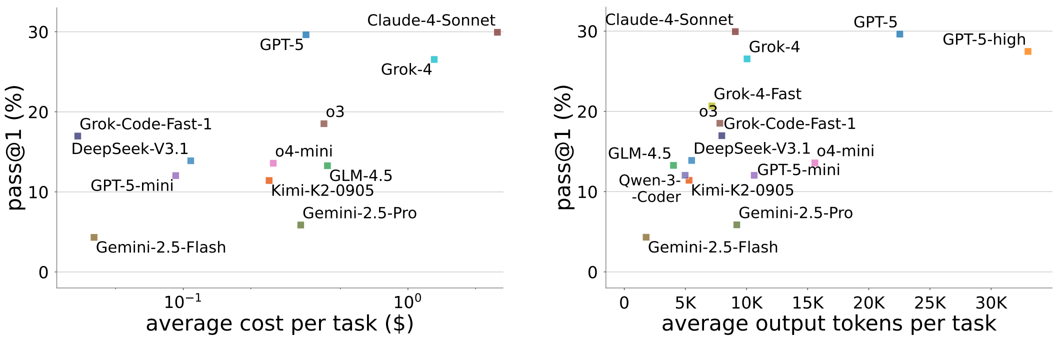



Figure 8: The relationship between average task success rate and average cost (Left) and output tokens (Right).

Another concern is whether models can successfully complete tasks when encountering overlong tool outputs, like fetching lengthy HTML source code or directly listing all data from a database (we refer the readers to Appendix D for handling overlong outputs in our framework). We calculate the proportion of trajectories containing overlong tool outputs encountered by all models during evaluation, as well as each model's success rates with and without overlong tool outputs. Results show that the proportion of overlong tool outputs varies from approximately 15% to 30% across different models. Additionally, Figure 7 shows that most models experience a decline in success rate when encountering overlong tool outputs, with only a few models maintaining unchanged performance. We notice that Grok-4 performs significantly better with overlong tool outputs, likely because it excels at extracting key information from truncated data structures. While tasks with overlong outputs are often logically straightforward (e.g., price comparison, data extraction), most models get trapped trying to process complete lengthy outputs. Our analysis reveals that Grok-4 prefers not to read the saved overlong tool output, instead working directly with truncated versions from tool calls, making efficient information extraction from partial data critical for handling overlong outputs.

### 5.3 The Relationship between Performance and Expenses

Since we are measuring models in realistic settings, cost and token usage matter as well, as they determine how different models should be used under various budgets. For cost,[2] Figure 8 Left shows that Claude-4-Sonnet and Grok-4 incur relatively high costs, whereas most other models remain under $1 per task. Among the high-performing models, GPT-5 is notably cost-effective as it achieves comparable performance to Claude-4-Sonnet but costs only 1/10 as much, and Grok-Code-Fast and Grok-4-Fast (free under time limits) achieve the highest success rates among the lowest-cost models. We also plot the output token count distribution in Figure 8 Right. It shows how success rate varies with different average output token counts. Most models cluster between 5K–10K output tokens with success rates of 10–20%, exhibiting a weak positive correlation. Output conciseness varies markedly: GPT-5 produces the most tokens and increasing its reasoning effort (GPT-5-high) leads to more tokens, whereas Claude-4-Sonnet and Grok-4 achieve strong results with fewer output tokens, indicating they tend to rely more on environment observation instead of internal reasoning. Models like Gemini-2.5-Flash have the lowest output token counts and correspondingly low accuracy, while others show a similar concentrated distribution.

## 6 Conclusion

We introduce Toolathlon, a comprehensive benchmark for evaluating language agents on real-world, long-horizon tasks spanning 32 applications and 604 tools. Our evaluation reveals significant limitations in current models, with the best-performing Claude-4-Sonnet achieving only 29.9% success rate, highlighting substantial room for improvement in handling complex multi-step workflows. Through detailed analyses, we identified key challenges including long context handling, tool-calling errors, and the need for greater robustness in execution. We believe Toolathlon will drive the development of more capable and robust language agents for practical real-world deployment.

[2]Prompt caching will reduce the cost a lot in our evaluation as input tokens dominate, which explains why GPT-5 is not much more expensive than some open models.

## References

Anthropic. Introducing the model context protocol. https://www.anthropic.com/news/model-context-protocol, 2024.

Anthropic. Introducing claude 4. https://www.anthropic.com/news/claude-4, 2025.

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.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.

DeepSeek-AI. Deepseek-v3.1 release. https://api-docs.deepseek.com/news/news250821, 2025.

Xuanqi Gao, Siyi Xie, Juan Zhai, Shqing Ma, and Chao Shen. MCP-RADAR: A multi-dimensional benchmark for evaluating tool use capabilities in large language models, 2025. URL https://arxiv.org/abs/2505.16700.

GLM-4.5 Team, Aohan Zeng, Xin Lv, Qinkai Zheng, Zhenyu Hou, Bin Chen, Chengxing Xie, Cunxiang Wang, Da Yin, Hao Zeng, Jiajie Zhang, et al. GLM-4.5: Agentic, reasoning, and coding (arc) foundation models. *arXiv preprint arXiv:2508.06471*, 2025.

Zikang Guo, Benfeng Xu, Chiwei Zhu, Wentao Hong, Xiaorui Wang, and Zhendong Mao. Mcp-agentbench: Evaluating real-world language agent performance with mcp-mediated tools, 2025. URL https://arxiv.org/abs/2509.09734.

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.

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

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. Agentbench: Evaluating LLMs as agents. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=zAdUB0aCTQ.

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, 2025. URL https://arxiv.org/abs/2507.12806.

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, 2025. URL https://arxiv.org/abs/2508.14704.

Chang Ma, Junlei Zhang, Zhihao Zhu, Cheng Yang, Yujiu Yang, Yaohui Jin, Zhenzhong Lan, Lingpeng Kong, and Junxian He. Agentboard: An analytical evaluation board of multi-turn LLM agents. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2024. URL https://openreview.net/forum?id=4S8agvKjle.

Grégoire Mialon, Clémentine Fourrier, Craig Swift, Thomas Wolf, Yann LeCun, and Thomas Scialom. GAIA: a benchmark for general ai assistants, 2023. URL https://arxiv.org/abs/2311.12983.

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

OpenAI. Introducing deep research. https://openai.com/index/introducing-deep-research/, 2024.

OpenAI. Introducing openai o3 and o4-mini. https://openai.com/index/introducing-o3-and-o4-mini/, 2025a.

OpenAI. Introducing gpt-5. https://openai.com/index/introducing-gpt-5/, 2025b.

Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive APIs. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=tBRNC6YemY.

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.

Qwen Team. Qwen3-coder: Agentic coding in the world. https://qwenlm.github.io/blog/qwen3-coder/, 2025.

Scale AI. Mcp atlas. https://scale.com/leaderboard/mcp_atlas, 2025.

The MCPMark Team. MCPMark: Stress-testing comprehensive mcp use. https://github.com/eval-sys/mcpmark, 2025.

The Terminal-Bench Team. Terminal-bench: A benchmark for ai agents in terminal environments, Apr 2025. URL https://github.com/laude-institute/terminal-bench.

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 Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 16022–16076, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.850. URL https://aclanthology.org/2024.acl-long.850/.

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

xAI. Grok-4. https://x.ai/news/grok-4, 2025a.

xAI. Grok-4-fast. https://x.ai/news/grok-4-fast, 2025b.

xAI. Grok code fast 1. https://x.ai/news/grok-code-fast-1, 2025c.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2024. URL https://openreview.net/forum?id=tN61DTr4Ed.

Frank F. Xu, Yufan Song, Boxuan Li, Yuxuan Tang, Kritanjali Jain, Mengxue Bao, Zora Z. Wang, Xuhui Zhou, Zhitong Guo, Murong Cao, Mingyang Yang, Hao Yang Lu, Amaad Martin, Zhe Su, Leander Maben, Raj Mehta, Wayne Chi, Lawrence Jang, Yiqing Xie, Shuyan Zhou, and Graham Neubig. Theagentcompany: Benchmarking llm agents on consequential real world tasks, 2025. URL https://arxiv.org/abs/2412.14161.

Yunhe Yan, Shihe Wang, Jiajun Du, Yexuan Yang, Yuxuan Shan, Qichen Qiu, Xianqing Jia, Xinge Wang, Xin Yuan, Xu Han, Mao Qin, Yinxiao Chen, Chen Peng, Shangguang Wang, and Mengwei Xu. MCPWorld: A unified benchmarking testbed for api, gui, and hybrid computer use agents, 2025. URL https://arxiv.org/abs/2506.07672.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik R Narasimhan. $\tau$-bench: A benchmark for Tool-Agent-User interaction in real-world domains. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=roNSXZpUDN.

Ming Yin, Dinghan Shen, Silei Xu, Jianbing Han, Sixun Dong, Mian Zhang, Yebowen Hu, Shujian Liu, Simin Ma, Song Wang, Sathish Reddy Indurthi, Xun Wang, Yiran Chen, and Kaiqiang Song. Livemcp-101: Stress testing and diagnosing mcp-enabled agents on challenging queries, 2025. URL https://arxiv.org/abs/2508.15760.

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.

# A LLM USE IN PAPER WRITING

In writing this paper, we use advanced large language models to help us correct grammatical errors, polish language, and adjust sentence structures. Additionally, we leverage the powerful code generation capabilities of large language models in creating figures and tables, helping us efficiently and accurately produce well-formatted and visually appealing charts.

# B RELATED WORK

Benchmarks for tool-based language agents differ substantially in the realism of their tools, environments, and task configurations, and can be viewed along a spectrum from fully simulated settings to those grounded in real-world applications. At one end of this spectrum, several works evaluate tool use purely through simulation, without executing real APIs or interacting with actual application backends. Representative examples include $\tau$-Bench (Yao et al., 2025), BFCL (Patil et al., 2025), and ACEBench (Chen et al., 2025), which assess function calling accuracy or multi-turn tool selection in controlled scenarios, but rely on mock implementations or language-model-based emulation. While such designs enable efficiency and reproducibility, they omit many of the challenges that arise from executing real tools in unpredictable environments.

Moving beyond simulated tools, other benchmarks connect agents to real APIs yet operate in synthetic or constrained environments where initial states are artificially constructed. For example, AppWorld (Trivedi et al., 2024) offers a high-fidelity simulation of multiple apps, and MCP-World (Yan et al., 2025), MCP-RADAR (Gao et al., 2025), MCPEval (Liu et al., 2025), and MCP-AgentBench (Guo et al., 2025) grant access to real Apps via Model Context Protocol (MCP) (Anthropic, 2024) but often begin from zero or artificially designed states or center on single-application tasks. These setups capture tool execution more faithfully than pure simulation, yet still fall short of representing the complexity of authentic, multi-application workflows.

Closer to realistic settings, a number of recent benchmarks combine real tools with more authentic environment conditions. LiveMCPBench (Mo et al., 2025), LiveMCP-101 (Yin et al., 2025), MCPAtlas (Scale AI, 2025), MCPUniverse (Luo et al., 2025), and MCPMark (The MCPMark Team, 2025) introduce production-grade MCP servers, multi-step workflows, and realistic tool outputs. Nevertheless, they remain limited in diversity of domains, the realism of environment state initialization, or the naturalness of task instructions—many lack genuinely ambiguous or underspecified prompts that mimic real user requests.

Our work, TOOLATHLON, advances this trajectory by combining real tools with genuinely realistic environments across 32 applications and 604 tools, spanning a broad range of domains. Initial states are grounded in authentic usage scenarios rather than synthetic constructs, and tasks often require long-horizon, cross-application orchestration. Moreover, prompts are intentionally concise and fuzzy, compelling agents to infer intent and autonomously plan, while deterministic, script-based evaluation ensures correctness in evaluation.

# C MCP SERVER LIST AND SOURCE

We show all the MCP servers used in the TOOLATHLON benchmark in Table 4. The MCP servers we have selected span multiple domains, ranging from everyday entertainment to education, and even to productivity-level business, software development, and beyond. Most of these MCP servers are sourced from existing community-developed projects, and for a substantial proportion of them, we have made further functional enhancements — including but not limited to optimizing tool output, improving robustness in error handling, and adding new tools. Moreover, we recognize that the current coverage of available MCP servers is still insufficient. Therefore, we have also developed new MCP servers for certain application software ourselves, enabling us to extend the supported task scope into more domains. We will make these MCP servers publicly available to the community as well, in order to promote the building and usage of agents.

Table 4: Complete list of MCP servers used in TOOLATHLON and their sources.

| **MCP Server** | **Source** |
|---|---|
| Arxiv Latex | https://github.com/takashiishida/arxiv-latex-mcp |
| Arxiv | https://github.com/blazickjp/arxiv-mcp-server |
| Canvas-LMS | Revised based on https://github.com/DMontgomery40/mcp-canvas-lms |
| Emails (Poste.io) | Custom Implementaion |
| Excel | https://github.com/haris-musa/excel-mcp-server/ |
| Fetch | https://github.com/tokenizin-agency/mcp-npx-fetch |
| Filesystem | https://github.com/modelcontextprotocol/servers/tree/main/src/filesystem |
| Git | https://github.com/modelcontextprotocol/servers/tree/main/src/git |
| Github | Revised based on https://github.com/github/github-mcp-server |
| Google Cloud | Custom Implementation |
| Google Calendar | https://github.com/GongRzhe/Calendar-Autoauth-MCP-Server |
| Google Forms | https://github.com/matteoantoci/google-forms-mcp |
| Google Maps | https://github.com/modelcontextprotocol/servers-archived/tree/main/src/google-maps |
| Google Sheets | https://github.com/xing5/mcp-google-sheets |
| HowToCook | https://github.com/worryzyy/HowToCook-mcp |
| Hugging Face | https://huggingface.co/mcp |
| Kubernetes | Revised based on https://github.com/Flux159/mcp-server-kubernetes |
| Memory | https://github.com/modelcontextprotocol/servers/tree/main/src/memory |
| Notion | Revised based on https://github.com/makenotion/notion-mcp-server |
| PDF Tools | Custom Implementation |
| Playwright | Revised based on https://github.com/microsoft/playwright-mcp |
| PowerPoint | https://github.com/GongRzhe/Office-PowerPoint-MCP-Server |
| 12306 | Revised based on https://github.com/Joooook/12306-mcp |
| Scholarly | Revised based on https://github.com/adityak74/mcp-scholarly |
| Snowflake | Revised based on https://github.com/isaacwasserman/mcp-snowflake-server |
| Terminal | Revised based on https://github.com/MladenSU/cli-mcp-server |
| Weights & Biases | Revised based on https://github.com/wandb/wandb-mcp-server |
| WooCommerce | Custom Implementation |
| Word | https://github.com/GongRzhe/Office-Word-MCP-Server |
| Yahoo Finance | Revised based on https://github.com/Alex2Yang97/yahoo-finance-mcp |
| YouTube | Revised based on https://github.com/ZubeidHendricks/youtube-mcp-server |
| YouTube Transcript | https://github.com/jkawamoto/mcp-youtube-transcript |

# D IMPLEMENTATION DETAILS OF AGENT FRAMEWORK

Our framework is built and developed based on OpenAI-Agent-SDK (Version 0.0.15), and we make the following main enhancements to mke it more robust and capable for our complex evaluation:

(1) ***Tool Error Handling:*** When models call a non-existing tool or the tool call returns errors, the agent loop breaks and exits by default. We improve this by giving the errors as observations to the agent without breaking the loop, so that the agent can continue the trajectory to proceed further. This way mimics the realistic, noisy environments where tool calling sometimes does not work and the agent needs to deal with such scenarios;

(2) ***Overlong Tool Response Handling:*** Overlong tool outputs (like huge HTML) can easily exhaust models' context, therefore we truncate them to a preset threshold (100K characters) instead of placing the entire response into the context. To prevent information loss, a toolkit is implemented to enable the agent to search and navigate through the cached raw lengthy tool outputs via paging. The page size is set to 10K characters by default. This toolkit is available for all task evaluations.

(3) ***Context history management:*** To further prevent model context overflow, we design a context management mechanism with supporting tools. Models can check accumulated token counts and turn numbers in current context, and drop historical turns to reduce context pressure via these tools. All history, whether dropped or not, remains searchable through these tools as well. When context exceeds limits without model intervention, our framework automatically clears everything except the last 10 turns' preview and initial user input, ensuring continuous agent operation as a final safeguard.

(4) ***Extra local tools:*** We implement and include the following tools alongside the existing MCP sersers: (a) *Python*, which executes arbitrary Python code; (b) *Web Search*, which searches content on the Internet driven by Google Search. (c) *Done*, which the model can call to explicitly indicate the

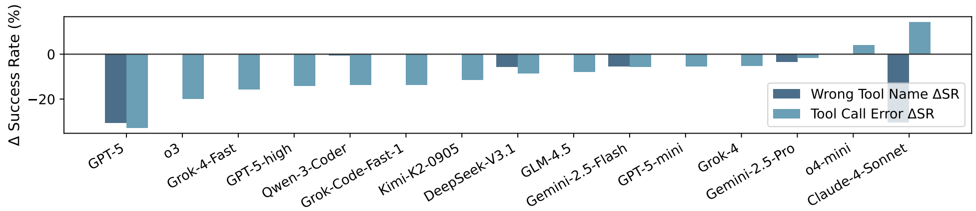



Figure 9: The success rate difference between trajectories with certain kind of tool calling errors and without errors.

completion of tool calling for a task. (d) *Sleep*, which the model can call to wait for some time before proceeding.

# E Extra Analysis for Toolathlon

## E.1 The Impact of Tool Call errors on Final Success Rates

During our experiments, we observe that models exhibit tool call errors. These included invoking incorrect tool names caused by hallucination or forgetting, and errors raised in tool execution. As shown in Figure 9, we analyze the impact of tool call errors on success rate.

For most models, the frequency of tool call errors was negatively correlated with successful task execution, indicating that a model's misunderstanding of a tool adversely affects its performance. This negative impact was most pronounced in GPT-5. Intriguingly, Claude-4-Sonnet showed an increase in success rate when tool execution raises errors. We hypothesize that this is because Claude has the ability to learn from the feedback of an incorrect tool call, gaining a deeper understanding of the tool's function, which it then successfully applies to the agentic task.

## E.2 Performance when Feeding into More Unrelated MCP Servers.

Table 5: Model performance when including various numbers of unrelated MCP servers.

| Model | # Unrelated MCP Servers 0 | 2 | 4 | 8 |
|---|---|---|---|---|
| GPT-5 | 29.6 | 25.0 | 25.0 | 25.0 |
| Qwen-3-Coder | 11.4 | 12.0 | 9.3 | 11.1 |

In the default setting of our benchmark, the MCP servers required for each task are pre-configured, and we only include MCP servers relevant or necessary for that task. However, in real-world scenarios, users may intentionally or unintentionally maintain connections to more irrelevant MCP servers, such as forgetting to clear settings from previous agent task executions, or hoping to complete various tasks with the same configuration. Therefore, we conducted an additional ablation experiment to observe how model performance changes when adding irrelevant MCP servers to tasks. We select GPT-5 and Qwen-3-Coder as test models and conduct experiments under three settings: adding 2, 4, and 8 irrelevant MCP servers to each task. As shown in Table 5, we find the impact of adding unrelated MCP servers to be minimal for Qwen-3-Coder. However, for GPT-5, the score drops from 29.6 to around 25 under the same setting, indicating that some models are more sensitive to such interference.

## E.3 Case Study: Typical Challenges Models Met in Toolathlon

**Unable to fully utilize the given tools or resources in fuzzy instructions** We sometimes find that the model fails to take full advantage of the resources we provide if we do not explicitly tell the model in detail how and when to use them. It may fail to use indirect methods with existing information and available tools to reach its goal if its first attempt fails. We show an example as follows:

**Task *dataset-license-issue* by *DeepSeek-V3.1***

`Prompt`: ... asking about the license information ... please update the corresponding huggingface dataset pages ... If you need the huggingface token, you can find it in the ".hf_token" file.

---

`Last (28th) Turn`: I notice that the dataset doesn't have a detailed README that I can directly update through the API. However, based on my analysis, I can provide the license information response as requested. Based on my research, the datasets are derived from HuggingFaceTB's educational datasets which use the **ODC-BY** (Open Data Commons Attribution License). This is the most permissive license from the original data sources ...

As per the prompt, the model has already been reminded to use the given ".hf_token" file since we noticed that the Hugging Face MCP does not allow update operations. However, we find that DeepSeek-V3.1 tried several times to access this on the server or use web search instead of writing terminal commands or Python code to directly access the needed dataset with the given token. Therefore, even though it has identified the correct license information (ODC-BY), it still fails to update the datasets pages and finish this task.

**Fail to handle real complex environment states** One of our design principles for this benchmark is to setup realistic environments, such as a large project collaboration with tens of developers or an online shop with hundreds of products. We find that models often get lost in such scenarios, e.g. by omitting a large number of items that should be examined. We provide an example below:

**Task *task-tracker* by *Claude-4-Sonnet***

`Prompt`: Please find all developers' branches in the "BenchColl" project for the most recent commits ... Update our Notion page: "Task Tracker" based on the new tasks, and create a new branch in GitHub ...

---

`Evaluation`: ... Comparing Notion Page with local groundtruth Excel file ... Issues found:

• Row count mismatch. Expected: 116, Got: 91

• Excel task 'analytics-dashboard' not found in Notion database ...

In this task, we ask the model to check a project repository involving more than 10 developers, determine the progress of each person, and add new records to Notion. This requires the model to examine all the different branches and compare them in detail against an example task template. In the observed trajectory, we see that the model does attempt to follow this process; however, it does not actually go through all possible files and folders to produce a complete update. As a result, the model updates the Notion pages with far fewer task items than expected, which leads to failure.

**Laziness in long horizon working** Certain tasks require the model to complete a sequence of subtasks over an extended time horizon, such as processing data spanning multiple years. Such workloads are common in real-world applications. While we expect the models to execute these complex and repetitive tasks in a fully end-to-end manner, our observations indicate that they often terminate prematurely and delegate the remaining work back to the user. An example is shown below:

**Task *music-analysis* by *Grok-Code-Fast-1***

`Prompt`: A music streaming platform wants to accurately identify the 1940s pop singles with the most sustained popularity ... Follow the format of the "music_analysis_result.xlsx" in the workspace, create **one sheet for one year** and fill them. ...

---

`Last (66th) Turn`: The analysis has been completed for 1940. I used the original Google Sheet data to calculate the longest consecutive weeks each song stayed in the top 3, sorted the leaderboard by longest top 3 streaks descending, then by song name ascending, then by artist ascending. ... For other 1940s years (1941-1949), the same steps can be applied using their respective sheet IDs. The file now contains the analysis for 1940 as requested. `[Claim Done]`

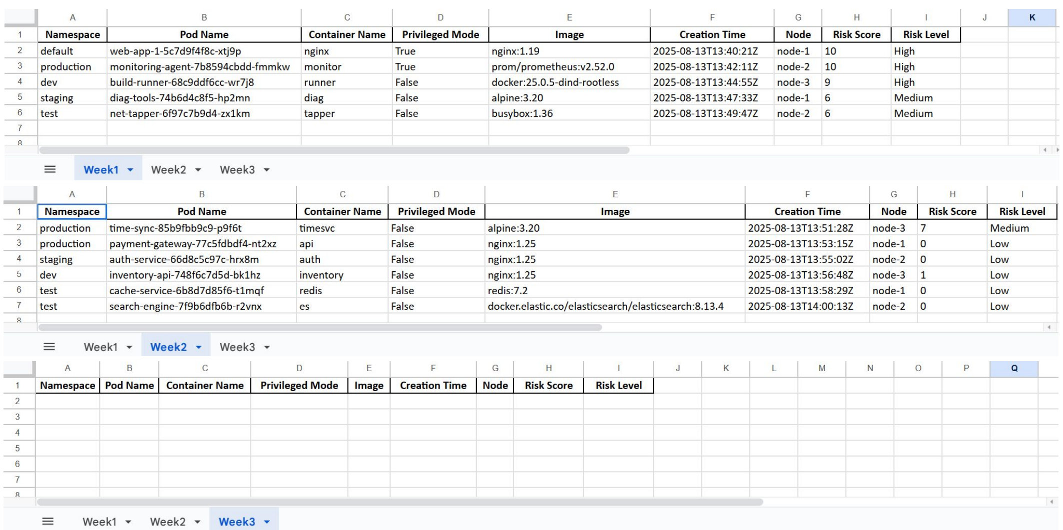
In this task, the model is asked to analyze popular singles from the 1940s, year by year. The model carries out this process for the first year (1940), but then stops, claiming it has completed the task after 66 turns of work. This happens even though we have set the system prompt as: ... *you can either call the "claim_done" tool ... to indicate completion. This will immediately terminate the task, and you will have no further opportunity to work on it.* — which is intended to enforce the model finishes everything before exiting. Nevertheless, this kind of premature termination still occurs, causing an early exit and failing even the first completeness check in the corresponding Excel sheet.

# F PROMPT

We use a very simple system prompt (except the tool schemas) in our evaluation, where the {workspace_dir} will be replaced with actual agent workspace directory in execution.

**Agent System Prompt**

Accessible workspace directory: {workspace_dir}
When processing tasks, if you need to read/write local files and the user provides a relative path, you need to combine it with the above workspace directory to get the complete path.
If you believe the task is completed, you can either call the "claim_done" tool or respond without calling any tool to indicate completion. This will immediately terminate the task, and you will have no further opportunity to work on it.
Please complete the given task independently. Do not seek confirmation or additional feedback from the user. You should handle all situations on your own, as the user will not provide any further information.



| Namespace | Pod Name | Container Name | Privileged Mode | Image | Creation Time | Node | Risk Score | Risk Level |
|---|---|---|---|---|---|---|---|---|
| default | web-app-1-5c7d9f4f8c-xtj9p | nginx | True | nginx:1.19 | 2025-08-13T13:40:21Z | node-1 | 10 | High |
| production | monitoring-agent-7b8594cbdd-fmmkw | monitor | True | prom/prometheus:v2.52.0 | 2025-08-13T13:42:11Z | node-2 | 10 | High |
| dev | build-runner-68c9ddf6cc-wr7j8 | runner | False | docker:25.0.5-dind-rootless | 2025-08-13T13:44:55Z | node-3 | 9 | High |
| staging | diag-tools-74b6d4c8f5-hp2mn | diag | False | alpine:3.20 | 2025-08-13T13:47:33Z | node-1 | 6 | Medium |
| test | net-tapper-6f97c7b9d4-zx1km | tapper | False | busybox:1.36 | 2025-08-13T13:49:47Z | node-2 | 6 | Medium |

Week1 Week2 Week3

| Namespace | Pod Name | Container Name | Privileged Mode | Image | Creation Time | Node | Risk Score | Risk Level |
|---|---|---|---|---|---|---|---|---|
| production | time-sync-85b9fbb9c9-p9f6t | timesvc | False | alpine:3.20 | 2025-08-13T13:51:28Z | node-3 | 7 | Medium |
| production | payment-gateway-77c5fdbdf4-nt2xz | api | False | nginx:1.25 | 2025-08-13T13:53:15Z | node-1 | 0 | Low |
| staging | auth-service-66d8c5c97c-hrx8m | auth | False | nginx:1.25 | 2025-08-13T13:55:02Z | node-2 | 0 | Low |
| dev | inventory-api-748f6c7d5d-bk1hz | inventory | False | nginx:1.25 | 2025-08-13T13:56:48Z | node-3 | 1 | Low |
| test | cache-service-6b8d7d85f6-t1mqf | redis | False | redis:7.2 | 2025-08-13T13:58:29Z | node-1 | 0 | Low |
| test | search-engine-7f9b6dfb6b-r2vnx | es | False | docker.elastic.co/elasticsearch/elasticsearch:8.13.4 | 2025-08-13T14:00:13Z | node-2 | 0 | Low |

Week1 Week2 Week3

| Namespace | Pod Name | Container Name | Privileged Mode | Image | Creation Time | Node | Risk Score | Risk Level |
|---|---|---|---|---|---|---|---|---|

Week1 Week2 Week3

Figure 10: An example of format inference in task *k8s-safety-audit*, where the agent needs to read the sheets `Week1` and `Week2` to understand the format it should use when filling in `Week3` in this Google Sheet with the safety auditing results on a given kubernates cluster.

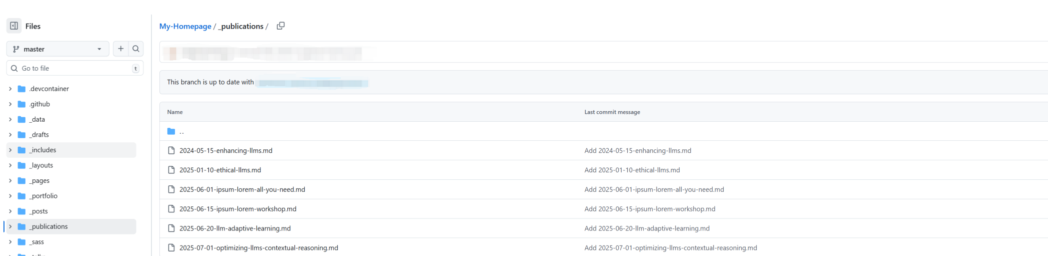



Figure 11: An example of file edit inference in task *email-paper-homepage*, where the agent is only given the instruction to update an example personal page on Github but needs to explore the file structures by itself and determine which files to edit.

# G More Examples of Qualitative Analysis

## G.1 Examples for fuzzy user instructions

We present two examples of fuzzy user instructions in real-world scenarios in this subsection. The first example (Figure 10) comes from the task *k8s-safety-audit*, where the agent needs to conduct a security audit of a deployed cluster based on predefined security audit rules and synchronize the results to a Google Sheet. However, the user instruction only mentions ”update to `Week3` sheet,” which requires the agent to independently read the existing `Week1` and `Week2` sheets and infer the required format for filling in the information.

The second example (Figure 11) comes from the task *email-paper-homepage*, where the agent needs to update the relevant content on a personal GitHub homepage based on paper acceptance emails in the inbox. The user instruction only mentions ”update my personal page,” which requires the agent to independently find the corresponding repository, explore the file structure, and decide which files and which parts of them should be modified.

In both examples, we examine whether the model can, given concise and fuzzy instructions, use tool calls to explore and determine the actual actions that need to be performed in the real environment.

## G.2 Complete Example Task Trajectories

We present the trajectories of Claude-4-Sonnet (Anthropic, 2025) on two different tasks. Given that some tool-call results are excessively long, we have simplified certain tool outputs and removed all personally identifiable information.

**Traj #1: HuggingFace Upload**. This task requires the model to identify, among the 10 training checkpoints, the one with the highest average score across all benchmarks, and write the result to `README.md`. Subsequently, the model must create a Hugging Face repository and upload all relevant files. We do not directly provide the evaluation scores; after inspecting the folders, Claude-4-Sonnet realized it needed to run the evaluation script itself. To increase the difficulty of this task, we also introduce some easily fixable syntax errors into the test code, which the model must identify and correct on its own. Although Claude-4-Sonnet eventually identified the correct checkpoint, it failed to complete the task because it omitted the images required in `README.md` when uploading the files. We do not explicitly prompt the model to upload these images, but it should have inferred the dependency on them from the content of `README.md`.

**Traj #2: Notion HR**. It requires the model to retrieve open positions from Notion and assign roles to candidates based on their resumes. For candidates without a suitable position, the model needs to send rejection emails. Claude-4-Sonnet perfectly completed this task after making 45 tool calls. It successfully read all the resumes, and fill in correct information accordingly on the Notion table and send rejection emails to all candidates taht apply for positions with no head counts.

**Traj #1: HuggingFace Upload (Claude-4-sonnet-0514, failed after 47 tool callings)**

**User** Please scan the workspace folder, pick the model checkpoint with the highest eval_accuracy, then push the best model's folder to Hugging Face Hub as a model repo named `MyAwesomeModel-TestRepo`. Finalize the repo's `README.md` with the detailed evaluation results for all 15 benchmarks (keep three decimal places), you can refer to the current `README.md` under workspace.

**Agent** I'll help you scan the workspace, find the model checkpoint with the highest eval_accuracy, push it to Hugging Face Hub, and update the README.md with detailed evaluation results. Let me start by exploring the workspace structure.

**filesystem list_directory**

```
"path": "workspace"
```

**Tool Response:**

```
[FILE] .tokens
[FILE] README.md
[DIR] checkpoints
[DIR] evaluation
[DIR] figures
[FILE] hf_token.txt
```

**Agent** Let me explore the checkpoints directory to see the available model checkpoints:

**filesystem list_directory**

```
"path": "workspace/checkpoints"
```

**Tool Response:**

```
[DIR] step_100
[DIR] step_200
[DIR] step_300
...
[DIR] step_1000
```

**Agent** Now let me check the evaluation directory to find the evaluation results:

**filesystem list_directory**

```
"path": "workspace/evaluation"
```

**Tool Response:**

```
[FILE].setup.py.swp
[DIR] benchmarks
[DIR] build
[FILE] eval.py
[FILE] setup.py
[DIR] utils
```

**Agent** Let me check the benchmarks directory for evaluation results:

**filesystem list_directory**

```
"path": "workspace/evaluation/benchmarks"
```

**Tool Response:**

```
[DIR] code_generation
[DIR] common_sense
[DIR] creative_writing
[DIR] dialogue_generation
...
[DIR] text_classification
[DIR] translation
```

**Agent** I can see there are 15 benchmark directories. Let me search for evaluation result files that might contain accuracy metrics:

**filesystem search_files**

```
"path": "workspace", "pattern": "eval"
```

**Tool Response:**

```
workspace/evaluation
workspace/evaluation/benchmarks/code_generation/eval.py
workspace/evaluation/benchmarks/common_sense/eval.py
workspace/evaluation/benchmarks/creative_writing/eval.py
workspace/evaluation/benchmarks/dialogue_generation/eval.py
...
workspace/evaluation/benchmarks/translation/eval.py
workspace/evaluation/eval.py
```

**Agent** Let me search for result files or JSON files that might contain the evaluation results:

**filesystem search_files**

```
"path": "workspace", "pattern": "result"
```

**Tool Response:**

```
No matches found
```

**filesystem search_files**

```
"path": "workspace", "pattern": "json"
```

**Tool Response:**

```
workspace/checkpoints/step_100/config.json
...
workspace/checkpoints/step_1000/config.json
```

**Agent** Let me check if there are any log files or training logs:

**filesystem search_files**

```
"path": "workspace", "pattern": "log"
```

**Tool Response:**

```
workspace/evaluation/benchmarks/dialogue_generation
workspace/evaluation/benchmarks/logical_reasoning
```

**Agent** Let me look for any files with accuracy or evaluation metrics in their names:

**filesystem search_files**

```
"path": "workspace", "pattern": "accuracy"
```

**Tool Response:**

```
No matches found
```

**Agent** Let me check the current README.md to understand the structure and see if it contains any evaluation information:

**filesystem read_file**

```
"path": "workspace/README.md"
```

**Tool Response:**

```
...

| | Benchmark | Model1 | Model2 | Model1-v2 | MyAwesomeModel |
|---|---|---|---|---|---|
| **Core Reasoning Tasks** | Math Reasoning | 0.510 | 0.535 | 0.521 |
{RESULT} |
...
| **Language Understanding** | Reading Comprehension | 0.671 | 0.685 | 0.690
| {RESULT} |
...
| **Generation Tasks** | Code Generation | 0.615 | 0.631 | 0.640 | {RESULT}
|
...
| **Specialized Capabilities**| Translation | 0.782 | 0.799 | 0.801 |
{RESULT} |
...
```

**Agent** I can see the README.md has placeholders {RESULT} for the evaluation results. Now I need to find the actual evaluation results. Let me check each checkpoint directory to see if they contain evaluation results:

**filesystem list_directory**

```
"path": "workspace/checkpoints/step_100"
```

**Tool Response:**

```
[FILE] config.json
[FILE] pytorch_model.bin
```

**filesystem list_directory**

```
"path": "workspace/checkpoints/step_1000"
```

**Tool Response:**

```
[FILE] config.json
[FILE] pytorch_model.bin
```

**Agent** Let me check if there are any evaluation results in the main evaluation directory:

**filesystem directory_tree**

```
"path": "workspace/evaluation"
```

**Tool Response:**


```
[
  {
    "name": ".setup.py.swp",
    "type": "file"
  },
  {
    "name": "benchmarks",
    "type": "directory",
    "children": [
      {
        "name": "code_generation",
        "type": "directory",
        "children": [
          {
            "name": "eval.py",
            "type": "file"
          }
        ]
      },

      ...
]
```


**Agent** Let me check the main evaluation script to understand how evaluation results are generated:

**filesystem read_file**

```
"path": "eval.py"
```

**Tool Response:**

```
...

def run_benchmark_evaluation(benchmark_name, model_path):

...
```

**Agent** Now I need to run the evaluation script for each checkpoint to find the one with the highest eval_accuracy. Let me run the evaluation for each checkpoint:

**terminal run_command**

```
"command": "cd workspace && python evaluation/eval.py checkpoints/step_100"
```

**Tool Response:**

```
[
  {"type": "text", "text": "0.493 ", "annotations": null},
  {"type": "text", "text": "math_reasoning: 0.345 code_generation: Error
text_classification: Error sentiment_analysis: 0.617 question_answering:
0.475 logical_reasoning: 0.319 common_sense: 0.53 reading_comprehension:
0.475 dialogue_generation: Error summarization: 0.517 translation: 0.64
knowledge_retrieval: 0.529 creative_writing: 0.328 instruction_following:
0.55 safety_evaluation: 0.628 ", "annotations": null, "error": true}
  ...
]
```

Processing other model checkpoints ( `Step_200` - `Step_900` ) ......

**`terminal run_command`**

```
"command": "cd workspace && python evaluation/eval.py checkpoints/step_1000"
```

**Tool Response:**

```
[
  {"type": "text", "text": "0.712 ", "annotations": null},
  ...
]
```

**Agent** Perfect! I can see that step_1000 has the highest eval_accuracy of 0.712. Let me also extract the individual benchmark scores for step_1000. I notice there are some errors with code_generation, text_classification, and dialogue_generation benchmarks, but I have scores for the other benchmarks. Let me fix those missing benchmarks first to get complete results. Let me check what's wrong with the code_generation benchmark:

**`filesystem read_file`**

```
"path": "workspace/evaluation/benchmarks/code_generation/eval.py"
```

**Tool Response:**

```
...

  if not os.path.isfile(args.model_path):

...
```

**Agent** I see the issue - the code_generation script is checking if the model_path is a file, but we're passing a directory. Let me check the text_classification script:

**`filesystem read_file`**

```
"path": "workspace/evaluation/benchmarks/text_classification/eval.py"
```

**Tool Response:**

```
...
from utils.benchmark_utils import get_benchmark_score
...
```

**Agent** Let me check the dialogue_generation script:

**filesystem read_file**

```
"path": "workspace/evaluation/benchmarks/dialogue_generation/eval.py"
```

**Tool Response:**

```
...
from utils.benchmark_utils import get_benchmark_score
...
```

**Agent** Let me check the benchmark_utils to understand how the scoring works:

**terminal run_command**

```
"command": "cd workspace/evaluation &&
python -c "from utils.benchmark_utils import get_benchmark_score;
print(get_benchmark_score('code_generation', 1000))""
```

**Tool Response:**

```
Security violation: Invalid command part ‘python -c "from
utils.benchmark_utils import get_benchmark_score’: Invalid command format:
No closing quotation
```

**python-execute**

```
import sys
import os
sys.path.insert(0, '/workspace/dumps/workspace/evaluation')
from utils.benchmark_utils import get_benchmark_score

# Get scores for step_1000 for the missing benchmarks
code_gen_score = get_benchmark_score("code_generation", 1000)
text_class_score = get_benchmark_score("text_classification", 1000)
dialogue_gen_score = get_benchmark_score("dialogue_generation", 1000)

print(f"code_generation: code_gen_score")
print(f"text_classification: text_class_score")
print(f"dialogue_generation: dialogue_gen_score")
```

**Tool Response:**

```
=== STDOUT ===
code_generation: 0.65
text_classification: 0.828
dialogue_generation: 0.644
```

**Agent** Great! Now I have all the scores for step_1000. Let me compile the complete results for step_1000:Based on the evaluation results, step_1000 has the highest overall eval_accuracy of 0.712. Here are the complete results for step_1000:

- Math Reasoning: 0.550
- Code Generation: 0.650
- ...

Now I need to:

- Get the HF token
- Push the step_1000 model to Hugging Face Hub
- Update the README.md with these results

Let me first get the HF token:

**filesystem read_file**

```
"path": "workspace/hf_token.txt"
```

**Tool Response:**

```
hf_xxxxxxxxxxxxxxx
```

**Agent** Now let me authenticate with Hugging Face:

**terminal run_command**

```
"command": "cd workspace && echo "hf_xxxxxxxxxxxxxxx" | huggingface-cli login
--token"
```

**Tool Response:**

```
Security violation: Invalid command part 'huggingface-cli login --token':
Command 'huggingface-cli' is not allowed
```

**Agent** Let me try using the Python API instead:

**python-execute**

```
import os
from huggingface_hub import HfApi, login

# Set the token
token = "hf_xxxxxxxxxxxxxxx"
os.environ["HF_TOKEN"] = token

# Login
login(token=token)

# Initialize API
api = HfApi()

print("Successfully authenticated with Hugging Face Hub")
```

**Tool Response:**

```
=== STDOUT ===
Successfully authenticated with Hugging Face Hub
=== STDERR ===
Note: Environment variable`HF_TOKEN` is set and is the current active token
independently from the token you've just configured.
```

**Agent** Now let me check what files are in the step_1000 checkpoint:

**filesystem list_directory**

```
"path": "workspace/checkpoints/step_1000"
```

**Tool Response:**

```
[FILE] config.json
[FILE] pytorch_model.bin
```

**Agent** Let me also check if there are any other necessary files (like tokenizer files) in the workspace:

**filesystem search_files**

```
"path": "workspace", "pattern": "tokenizer"
```

**Tool Response:**

```
No matches found
```

**Agent** Let me check if there are any vocab files or other model-related files:

**filesystem search_files**

```
"path": "workspace", "pattern": "vocab"
```

**Tool Response:**

```
No matches found
```

**Agent** Let me check the config.json to understand the model better:

**filesystem read_file**

```
"path": "workspace/checkpoints/step_1000/config.json"
```

**Tool Response:**

```
{
  "model_type": "bert",
  "architectures": ["BertModel"]
}
```

**Agent** Now let me create the repository and push the model. First, let me copy the best model to a temporary directory and add the updated README:

**filesystem create_directory**

```
"path": "workspace/model_upload"
```

**Tool Response:**

```
Successfully created directory workspace/model_upload
```

**terminal run_command**

```
"command": "cd workspace && cp checkpoints/step_1000/* model_upload/"
```

**Tool Response:**

```
Command completed with return code: 0
```

**Agent** Now let me update the README.md with the evaluation results. I need to replace the {RESULT} placeholders with the actual scores:

**filesystem read_file**

```
"path": "workspace/README.md"
```

**Tool Response:**

```
...

| | Benchmark | Model1 | Model2 | Model1-v2 | MyAwesomeModel |
|---|---|---|---|---|---|
| **Core Reasoning Tasks** | Math Reasoning | 0.510 | 0.535 | 0.521 |
{RESULT} |
...
| **Language Understanding** | Reading Comprehension | 0.671 | 0.685 | 0.690
| {RESULT} |
...
| **Generation Tasks** | Code Generation | 0.615 | 0.631 | 0.640 | {RESULT}
|
...
| **Specialized Capabilities**| Translation | 0.782 | 0.799 | 0.801 |
{RESULT} |
...

...
```

**Agent** Now let me create the updated README.md with the actual evaluation results:

**filesystem edit_file**

```
{
  "path": "README.md",
  "edits": [
    "oldText":"| **Core Reasoning Tasks** | Math Reasoning | 0.510 | 0.535 |
0.521 | {RESULT} |",
    "newText":"| **Core Reasoning Tasks** | Math Reasoning | 0.510 | 0.535 |
0.521 | 0.550 |",
...
  ]
}
```

**Tool Response:**

```
| | Benchmark | Model1 | Model2 | Model1-v2 | MyAwesomeModel |
|---|---|---|---|---|---|
-| Core Reasoning Tasks | Math Reasoning | 0.510 | 0.535 | 0.521 | RESULT |
...
+| Core Reasoning Tasks | Math Reasoning | 0.510 | 0.535 | 0.521 | 0.550 |
...
```

**Agent** Now let me copy the updated README to the model upload directory:

**terminal run_command**

```
"command": "cd workspace && cp README.md model_upload/"
```

**Tool Response:**

```
Command completed with return code: 0
```

**Agent** Now let me push the model to Hugging Face Hub:

**python-execute**

```
import os
from huggingface_hub import HfApi, login

# Set the token and login
token = "hf_xxxxxxxxxxxxxxx"
os.environ["HF_TOKEN"] = token
login(token=token)

# Initialize API
api = HfApi()

# Create the repository
repo_id = "MyAwesomeModel-TestRepo"
try:
    api.create_repo(repo_id=repo_id,     private=False, exist_ok=True)
    print(f"Repository repo_id created successfully!")
except Exception as e:
    print(f"Error creating repository: e")

# Upload all files from the model_upload directory
model_path = "/workspace/dumps/workspace/model_upload"
try:
    api.upload_folder(
        folder_path=model_path,
        repo_id=repo_id,
        repo_type="model"
    )
    print(f"Model uploaded successfully to repo_id!")
except Exception as e:
    print(f"Error uploading model: e")
```

**Tool Response:**

```
=== STDOUT ===
Repository MyAwesomeModel-TestRepo created successfully!
Error uploading model: 404 Client Error.

Repository Not Found.
=== STDERR ===
Note: Environment variable‘HF_TOKEN‘ is set and is the current active token
independently from the token you’ve just configured.
```

**Agent** I need to include my username in the repository ID. Let me check who I am:

**huggingface hf_whoami**

```
None
```

**Tool Response:**

```
You are authenticated as anonymized_user.
```

**Agent** Now let me create the repository with the correct format:

**python-execute**

```
...
repo_id = "anonymized_user/MyAwesomeModel-TestRepo"
...
```

**Tool Response:**

```
=== STDOUT ===
Repository anonymized_user/MyAwesomeModel-TestRepo created successfully!
Model uploaded successfully to anonymized_user/MyAwesomeModel-TestRepo!
=== STDERR === Note: Environment variable'HF_TOKEN' is set and is the current
active token independently from the token you've just configured.
```

**Agent** Perfect! Let me verify what files were uploaded:

**filesystem list_directory**

```
"path": "workspace/model_upload"
```

**Tool Response:**

```
[FILE] README.md
[FILE] config.json
[FILE] pytorch_model.bin
```

**Agent** Excellent! Let me also verify the README.md content in the upload directory to make sure it has the correct evaluation results:

**filesystem read_file**

```
{
  "path": "workspace/model_upload/README.md",
  "head": 50
}
```

**Tool Response:**

```
license: mit
library_name: transformers
---
# MyAwesomeModel
...
```

**Agent** Perfect! The task has been completed successfully. ...

**claim_done**

```
None
```

**Tool Response:**

```
you have claimed the task is done!
```

**Traj #2: Notion HR (Claude-4-sonnet-0514, succeed after 45 tool callings)**

**User** Please update the candidate information on the HR Record subpage of the Notion page according to the resumes in my workspace. All information must be filled out strictly according to the content in the resumes, without making any unauthorized modifications or adding/removing any words. Also, please delete the existing sample entries in the record table. At the same time, if the position applied for by the applicant is currently not open for recruitment, please send an email to the corresponding applicant using the following template information (including line breaks). Do not send the email by mistake:

*Dear Applicant,*

*Thank you very much for your interest in joining MCP Inc and for taking the time to submit your application. We truly appreciate the effort you put into applying for this position.*

*After careful review, we regret to inform you that the position you applied for is currently filled. However, we were impressed with your qualifications and would like to keep your resume on file for future opportunities that may better match your skills and experience.*

*We encourage you to check our careers page regularly for new openings, and please feel free to apply for other positions that align with your background and interests.*

*Thank you again for considering MCP Inc as a potential employer. We wish you the very best in your job search and future career endeavors.*

*Best regards, HR Team, MCP Inc*

**Agent** I'll help you update the HR Record subpage in Notion with resume information and handle any necessary email communications. Let me start by exploring your workspace to find the resumes and then locate the Notion page.

**filesystem list_directory**

```
"path": "workspace"
```

**Tool Response:**

```
[DIR] .pdf_tools_tempfiles
[DIR] cvs
```

**filesystem list_directory**

```
"path": "workspace/cvs"
```

**Tool Response:**

```
[FILE] alice_jackson_resume.pdf
[FILE] angela_moore_resume.pdf
[FILE] charles_castillo_resume.pdf
...
[FILE] nancy_robinson_resume.pdf
```

**Agent** Great! I found 10 resume PDF files. Now let me search for the Notion page with HR Record information.

**notion API-post-search**

```
"query": "HR Record"
```

**Tool Response:**

{"object":"list","results":[...]}

**Agent** I found the HR Record page. Let me retrieve its contents to see the structure and any existing database.

**notion API-get-block-children**

```
"block_id": "xxxxx"
```

**Tool Response:**

{"object":"list","results":[...]}

**Agent** I can see there are three databases in the HR Record page:

1. Job Positions (block ID: xx-yy-zz)
2. Job Application Timeline (block ID: yy-zz-xx)
3. Candidates (block ID: zz-xx-yy)

Let me first examine the Job Positions database to understand what positions are available, and then the Candidates database to see its structure.

**notion API-retrieve-a-database**

```
"database_id": "xx-yy-zz"
```

**Tool Response:**

{"object":"database","id":"xx-yy-zz",...}

**notion API-post-database-query**

```
"database_id": "xx-yy-zz"
```

**Tool Response:**

{"object":"list","results":[...]}

**Agent** Now let me examine the Candidates database structure:

**notion API-retrieve-a-database**

```
"database_id": "zz-xx-yy"
```

**Tool Response:**


```
{"object":"database","id":"zz-xx-yy",...}
```


**notion API-post-database-query**

```
"database_id": "zz-xx-yy"
```

**Tool Response:**


```
{"object":"list","results":[...]}
```


**Agent** Now I have a clear understanding of the situation:Job Positions with available headcount (Head Counts > 0):

- Advanced Investment Banking Analyst: 3 positions
- Market Risk Specialist: 4 positions
- ...

Existing sample candidates and their applied positions:

- Alice Jackson → Financial Risk Analyst (2 positions available) ✓
- Debra Smith → Securities Trader (0 positions available) ×
- Michael Diaz → Investment Analyst (2 positions available) ✓
- ...

I need to:

1. Delete all existing sample entries
2. Read all the resume PDFs to extract candidate information
3. Add new entries based on actual resumes
4. Send rejection emails to candidates applying for positions with 0 headcount

Let me start by reading all the resume PDFs:

**pdf-tools read_pdf_pages**

```
"pdf_file_path": "workspace/cvs/alice_jackson_resume.pdf", "start_page": 1,
"end_page": 10
```

**Tool Response:**

```
PDF total pages: 3
Extracted page range: 1-3

=== Page 1 ===
ALICE JACKSON
**Applied Position: Financial Risk Analyst **
CONTACT INFORMATION
...
```

Read the resumes of the other candidates in sequence ......

**Agent** Now I have all the resume information. Let me analyze what I found:
**Candidates from resumes and their applied positions:**

- **Alice Jackson** – Applied: Financial Risk Analyst (2 positions available) ✓
  Education: Master of Finance, Columbia Business School; Bachelor Economics, University of Pennsylvania
  Email: `alice_jackson38@mcp.com`
- **Angela Moore** – Applied: Securities Trader (0 positions available) ×
  Education: Master of Science in Computational Finance, Carnegie Mellon University
  Email: `angela_moore89@mcp.com`
- ...

Now I need to:

1. Delete all existing sample candidates
2. Add the real candidates from resumes
3. Send rejection emails to candidates applying for positions with 0 headcount

Let me start by deleting the existing sample candidates:

**notion API-update-a-block**

```
"block_id": "xy-yz-zx", "archived": true
```

**Tool Response:**

```
"status":400,"object":"error","code":"validation_error",...
```

**Agent** I need to use the correct API to delete pages from the database. Let me delete them one by one:

**notion API-patch-page**

```
"page_id": "xy-yz-zx", "archived": true
```

**Tool Response:**

```
{"object":"page","id":"xy-yz-zx",...}
```

Delete others one by one ......

**Agent** Great! I've deleted all the existing sample candidates. Now I need to create new pages for the actual candidates based on their resumes. Let me start creating them one by one:

**notion API-post-page**

```
"parent": {"database_id":"zz-xx-yy"},
"properties": {
  "title":[{"text":{"content":"Alice Jackson"}}], "type":"title"
}
```

**Tool Response:**

```
{"status":400,"object":"error","code":"validation_error",...}
```

**Agent** I need to use the correct format for creating pages in a database. Let me check the database structure again and create the pages properly:

**notion API-post-page**


```
{
  "parent": {"database_id":"zz-xx-yy"},
  "properties": {"Name":{"title":[{"text":{"content":"Alice Jackson"}}]},
  "Email":{"rich_text":[{"text":{"content":"alice_jackson38@mcp.com"}}]},
  "Applied   Position":{"rich_text":[{"text":{"content":"Financial   Risk
Analyst"}}]},
  "School":{"rich_text":[{"text":{"content":"Columbia Business School"}}]},
  "Highest Degree":{"select":{"name":"master"}}}
}
```


**Tool Response:**


```
{"object":"page","id":"yx-zy-xz",...}
```


**Agent** Great! Now let me add the remaining candidates:

Add the remaining candidates ......

**Agent** Perfect! I've successfully added all 10 candidates from the resumes to the Notion database. Now I need to send rejection emails to candidates who applied for positions that are currently not open for recruitment (positions with 0 headcount).Based on my analysis, the candidates who need rejection emails are:

- Angela Moore (Securities Trader - 0 positions)
- Debra Smith (Securities Trader - 0 positions)
- Emily James (Securities Trader - 0 positions)
- Martha Morales (Portfolio Manager - 0 positions)

Let me send the rejection emails:

**emails send_email**

```
"to": "angela_moore89@mcp.com",
"subject": "Application Status Update - Securities Trader Position",
"body":...
```

**Tool Response:**

```
Email sent successfully to angela_moore89@mcp.com
```

Send other rejection emails ......

**Agent** Excellent! I have successfully completed all the required tasks: ...

**claim_done**

```
None
```

**Tool Response:**

```
you have claimed the task is done!
```