Agent-Diff: Benchmarking LLM Agents on Enterprise API Tasks via Code Execution with State-Diff-Based Evaluation

Hubert M. Pysklo* **and Artem Zhuravel*** **and Patrick D. Watson**
Minerva University
San Francisco, California, USA
hubert@uni.minerva.edu, zhuravel@uni.minerva.edu, pwatson@minerva.edu

Abstract

We present Agent-Diff, a novel benchmarking framework for evaluating agentic Large Language Models (LLMs) on real-world productivity software API tasks via code execution. Agentic LLM performance varies due to differences in models, external tool access, prompt structures, and agentic frameworks. Benchmarks must make fundamental trade-offs between a sandboxed approach that controls for variation in software environments and more ecologically valid approaches employing real services. Agent-Diff attempts to capture the desirable features of both of these approaches by including access to the real API interfaces for software services while sandboxing the environment in which calls are made, processed, and evaluated. This approach relies on two key innovations. The first is a novel state-diff contract - a declarative specification of the expected before/after state change - which separates process from outcome: rather than fuzzy trace or parameter matching, we define task success as whether the expected change in environment state was achieved. The second is a novel sandbox built on containerized replicas of enterprise APIs, allowing all models to interact with the same service interfaces through code execution. This enables controlled evaluation against a common set of state-diff contracts while preserving the structure of real-world API interaction. Using the Agent-Diff framework, we provide benchmarks for nine LLMs across 224 tasks utilizing enterprise software workflows. In addition, we evaluate the robustness of the framework with ablation experiments to assess the contribution of access to API documentation on benchmark performance.
Code and data: https://github.com/agent-diff-bench/agent-diff.

[1]Equal contribution.

1 Introduction

Agentic LLMs (Yao et al., 2023a) actively interact with external software environments (Masterman et al., 2024) and are capable of code generation and code-centric reasoning (Deng et al., 2025b; Anthropic, 2026b; Google, 2026; Cursor, 2026). While the dominant paradigm for agent–environment interaction remains tool calling through structured schemas such as the Model Context Protocol (Anthropic, 2024), such approaches face scalability constraints as tool catalogs grow into the hundreds or thousands of definitions (Gao et al., 2025; Luo et al., 2025). As such, rather than relying on rigid tool definitions, agents can write and execute code to interact with APIs directly (Jones and Kelly, 2025)–an approach that scales more efficiently but demands that agents plan, iterate, and maintain state over extended action sequences. Current benchmarks do not adequately measure these capabilities in enterprise productivity settings–an increasingly common agentic application domain (Anthropic, 2026a).

Existing benchmarks occupy different points in a shared evaluation design space. In particular, they differ along the following axes:

- **Interaction model:** structured tool calling via MCP/JSON schemas (Gao et al., 2025; Wang et al., 2025; Luo et al., 2025), direct API access (Li et al., 2023), or agent-written code (Deng et al., 2025a; Merrill et al., 2026).
- **Environment:** synthetic or mocked settings (Gao et al., 2025; Maekawa et al., 2025), live services (Wang et al., 2025; Luo et al., 2025), or reproducible sandboxes (Yan et al., 2025; Deng et al., 2025a).
- **Evaluation signal:** trace matching (Gao et al., 2025), outcome or state verification (Yan et al., 2025; Yao et al., 2024), or judge-based scoring (Wang et al., 2025).
- **Observability assumptions:** white-box instru-

mentation (Yan et al., 2025) versus black-box APIs (Wang et al., 2025; Luo et al., 2025).

- **Task horizon:** the minimum number of actions required under an optimal policy, which varies substantially across benchmarks and affects how much multi-step planning and state tracking an agent must perform.

The current work introduces Agent-Diff, a benchmarking framework that combines desirable features of both live and sandboxed approaches: agents interact with real API interfaces of enterprise productivity services (Slack, Box, Linear, Google Calendar) inside isolated environments that can be instantiated identically across runs. This approach helps maintain ecological validity while maintaining reproducibility and environmental control.

Key to this method is evaluation methodology– a state-diff contract that decouples processes and outcome measures. Rather than fuzzy trace or parameter matching, we compute the state difference between sandbox snapshots at the beginning and the end of task execution, and define task success by whether the expected change in environment state was achieved. Because diffs are computed over the full environment state, we can enforce invariants and detect unintended side effects (e.g., modifications or deletions of unrelated resources). A unified sandbox built on containerized API replicas ensures that all models interact with identical service interfaces under controlled conditions, enabling rigorous comparison across agents.

We use these state-diff contracts to evaluate models across a novel benchmark suite comprising 224 tasks. These tasks include single- and multi-step actions that orchestrate workflows across scheduling, messaging, file management, and project management.

2 Related Work

A core trade-off in agent evaluation is between ecological fidelity and reproducibility. Benchmarks that evaluate agents against mocked or synthetic tool environments (Li et al., 2023; Gao et al., 2025; Maekawa et al., 2025) gain determinism but can underspecify real enterprise API behavior (Bandlamudi et al., 2025). Benchmarks that connect to live services (Wang et al., 2025; Luo et al., 2025) increase realism but introduce temporal instability, requiring dynamic ground-truth retrieval. Bandlamudi et al. (Bandlamudi et al., 2025) bridge this gap by comparing agent API responses against ground-truth executions, but their response diffing validates only return values and not whether the intended state change was achieved or whether an unrelated state was inadvertently modified.

Many evaluation protocols validate how an agent acted rather than what it accomplished. Trace-based scoring (Gao et al., 2025; Maekawa et al., 2025) – matching an expected tool-call sequence – is brittle to equivalent alternative strategies and does not guarantee that the intended state change occurred. τ-Bench (Yao et al., 2024) advances evaluation by comparing database state against annotated goals in simulated retail and airline domains, but checks only the target state – not whether unrelated state was inadvertently modified. Reliance on LLM-as-a-judge (Wang et al., 2025) introduces further risk: such judges can be non-deterministic and biased toward stylistic features rather than correctness (Luo et al., 2025). Approaches that instrument application internals (Yan et al., 2025) improve observability but presume open-source or modifiable systems, misaligned with closed-source enterprise SaaS constraints.

Agent-Diff addresses these limitations simultaneously. Our containerized replicas of production APIs provide ecological fidelity with full reproducibility, without the temporal instability of live services or the oversimplification of mocked environments. Deterministic state diffing evaluates what changed rather than how, capturing both intended outcomes and unintended side effects, unlike goal-state comparison, response diffing, or trace matching. Finally, whereas prior code-execution benchmarks primarily focus on software-engineering tasks in repository environments, we evaluate code-executing agents on black-box enterprise API workflows.

3 Agent-Diff

Agent-Diff evaluates code-executing agents on long-horizon enterprise API tasks through state changes in the environment. It rests on three core requirements: (1) black-box equivalence, (2) state-centric modelling, and (3) deterministic isolation.

3.1 Agent-Diff Environments

Our design hypothesis is that closed-source enterprise APIs can be replicated with sufficient fidelity to produce representative evaluation scores because the contract – not the implementation – defines cor-

rectness from an agent's perspective.

Consider an API as a function $f : (r, S) \to (S', \mathcal{O})$, mapping a request r and a state S to an updated state S' and some output $\mathcal{O}$. An agent observes only r and $\mathcal{O}$; the internal path from S to S' is opaque.

Define the set of all possible requests that can be made to an API as $\mathcal{R} := \{r_1, r_2, \dots\}$, and assume that $\mathcal{R}^* \subset \mathcal{R}$ is the set of requests exercised by the benchmark tasks. We say that two APIs f and $\hat{f}$ are behaviorally equivalent over $\mathcal{R}^*$ if:

$$\forall\, r \in \mathcal{R}^*,\, S :\; f(r, S) = \hat{f}(r, S) \tag{1}$$

Full equivalence over all possible requests $\mathcal{R}$ is hard to achieve and not necessary. Therefore, we target equivalence over $\mathcal{R}^*$. Because the APIs in our benchmark are predominantly CRUD interfaces over typed entities, internal side effects (notifications, analytics, caching) are invisible to the agent and irrelevant for evaluation – our replicas need only produce equivalent outputs for equivalent input.

Crucially, all information required to construct $\hat{f}$ is publicly available: entity schemas (via OpenAPI or GraphQL introspection), operation semantics, response formats (via official SDKs), and error conditions (Slack Technologies, 2026b; Box, Inc., 2026a; Google LLC, 2026; Linear Inc., 2026a). We validate replica fidelity through automated conformance testing against production APIs and adversarial edge-case probing (Appendix A.12).

State-Centric Modeling. We model each service as a state machine whose state S is a typed relational database derived from public API specifications. Entity types (e.g., `User`, `Channel`, `Message` in Slack; `File`, `Folder` in Box) map directly to tables, and all operations reduce to three transitions:

$$\textsc{Insert} : S \to S \cup \{x\} \tag{2}$$

$$\textsc{Update} : S \to (S \setminus \{x\}) \cup \{x'\} \tag{3}$$

$$\textsc{Delete} : S \to S \setminus \{x\}, \tag{4}$$

where x is an entity and x' is an updated entity. Instead of validating API call traces, we snapshot tables before and after execution ($T_{\text{before}}, T_{\text{after}}$) and compute the state diff ($\Delta_{\text{add}}, \Delta_{\text{del}}, \Delta_{\text{mod}}$):

$$\Delta_{\text{add}} = T_{\text{after}} \setminus T_{\text{before}} \tag{5}$$

$$\Delta_{\text{del}} = T_{\text{before}} \setminus T_{\text{after}} \tag{6}$$

$$\Delta_{\text{mod}} = \{(x, x') : x.\text{pk} = x'.\text{pk} \wedge x \neq x'\}, \tag{7}$$

where x.pk is the primary key of the original entity and x'.pk is the primary key of the updated entity. The resulting state diff is compared against the task's expected state changes. These diffs are computed over the task-local sandbox state rather than over production-scale enterprise databases, which keeps full-state diffing tractable. This evaluation paradigm is best suited to tasks whose success can be represented as persistent changes to structured state, such as creating, updating, deleting, tagging, or moving enterprise objects. Tasks whose ground truth is primarily a returned value, subjective preference, visual quality, or open-ended reasoning may require response-based checks, human evaluation, or LLM-as-a-judge grading instead.

Determinism and Isolation. Each benchmark run executes in a fresh environment. We use PostgreSQL schema isolation so that every environment operates in its own namespace (e.g., `state_{env_id}`), enabling concurrent execution without cross-contamination. Environments are seeded from pre-defined templates, ensuring identical initial states across runs ρ. This architecture guarantees that for any task τ and agent $\mathcal{A}$:

$$\forall \rho_i, \rho_j \in \text{Runs}(\tau, \mathcal{A}) : S_0^{(\rho_i)} = S_0^{(\rho_j)}, \tag{8}$$

where $S_0^{(\rho_i)}$ and $S_0^{(\rho_j)}$ are the states at the beginning of the runs ρ_i and ρ_j, accordingly. This enables reproducible evaluation without the temporal instability of live APIs, and ensures that the observed state diff reflects only the actions of the evaluated agent.

3.2 Agent-Diff Benchmark

Task taxonomy. Each task τ is characterized along five dimensions (Table 2). The primary axis is task horizon $n^*(\tau)$: the minimum number of endpoint invocations required under an optimal policy, directly controlled via the sampling procedure described below.

Two structural dimensions emerge from the sampled endpoint multiset:

- **Operation profile**: the set of API primitives required – search, create, read, update, and/or delete (non-exclusive).
- **Entity scope** d_{scope}: single-entity vs. multi-entity state changes.

Two pragmatic dimensions are shaped during generation and human curation to operationalize

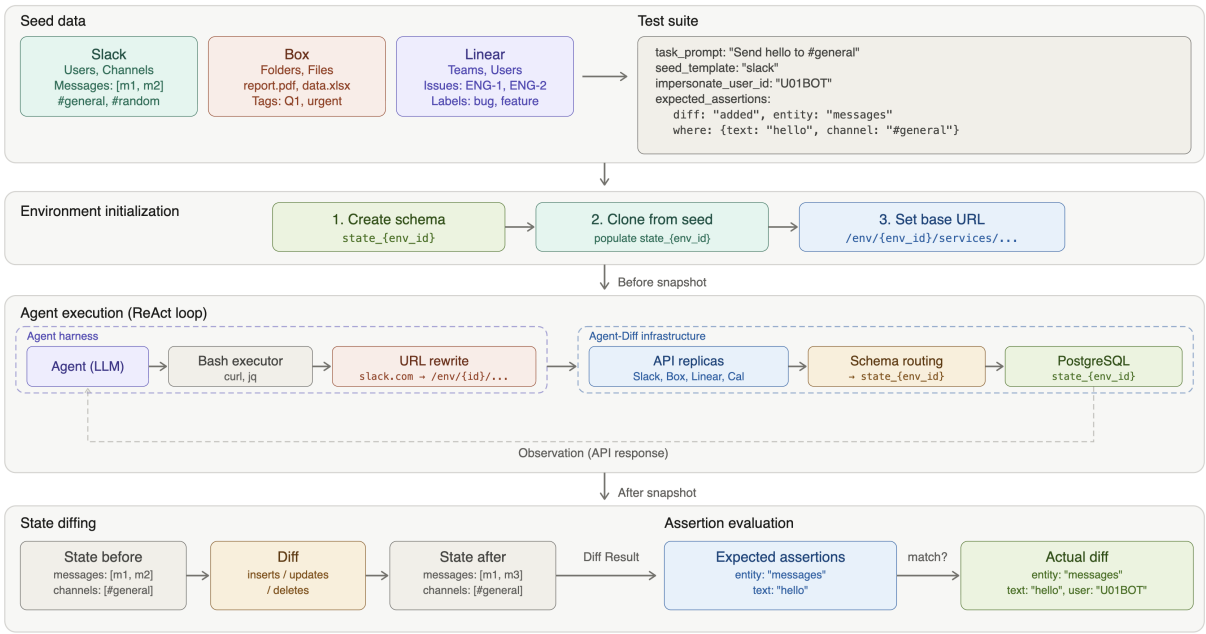

Figure 1: End-to-end sandbox architecture. The agent emits code (Bash/Python) that executes inside a container. All network traffic is intercepted and routed to containerized API replicas backed by per-environment PostgreSQL schemas. Entity tables are snapshotted to produce the `DiffResult` used for evaluation. Appendix A.10.

the problem space beyond the trace-level tool calling:

- **Information availability** d_{info}: whether required identifiers are provided explicitly in the prompt or must be discovered via API queries. This is governed by the identifier policy in the generation prompt.
- **Ambiguity** d_{amb}: the degree to which the prompt underspecifies the exact target. Low-ambiguity tasks are produced by the LLM; medium- and high-ambiguity variants are introduced during human curation (e.g., removing explicit identifiers, adding distractor entities).

Tool-call budget. We target a heavy-tailed distribution of task horizons by sampling intended tool-call counts from a shifted truncated exponential over $\{1, \ldots, 40\}$:

$$\Pr[n = k] \propto e^{-\lambda(k-1)}, \quad k \in \{1, \ldots, 40\}. \quad (9)$$

After feasibility filtering and manual curation, the realized distribution has mean $\bar{n}^* = 5.3$ with range 1–24, reflecting a slight upward shift from the rejection of trivially simple proposals.

Endpoint sampling. For each service s (Slack, Box, Linear, Google Calendar), let $\mathcal{E}_s$ denote its supported endpoints. Conditioned on s and target horizon n, we sample an intended endpoint multiset $(\varepsilon_1, \ldots, \varepsilon_n)$ i.i.d. uniformly from $\mathcal{E}_s$:

$$\Pr[\varepsilon_i = \varepsilon] = \frac{1}{|\mathcal{E}_s|}, \qquad \varepsilon \in \mathcal{E}_s. \quad (10)$$

Uniform sampling encourages coverage across the API surface. For example, a Calendar task with target horizon three might sample {`events.list`, `freebusy.query`, `events.insert`}, corresponding to a workflow that inspects calendar state, checks availability, and creates an event. The multiset guides task generation rather than defining a required trace; task success is determined by whether the final state diff satisfies the task assertions. Incoherent samples are removed during feasibility filtering and human curation. The benchmark spans 108 unique endpoints (Box: 27, Slack: 25, Linear: 19, Calendar: 37).

Task generation pipeline. For each sampled endpoint multiset, we use a mix of LLMs (Claude Opus 4.5 and Gemini 3) to generate: (1) a natural-language user prompt whose completion induces state changes consistent with $(\varepsilon_1, \ldots, \varepsilon_n)$, (2) a deterministic seed template defining the initial sandbox state S_0, and (3) a set of declarative assertions describing the expected state diff and forbidden side effects. The endpoint multiset determines the structural dimensions of the task – its operation

Table 1: Representative task spanning the difficulty spectrum. n^*: minimum API calls.

Svc	Task Prompt	n^*
Box	Search for all plain-text files about Argentina's 2001 economic crisis. You should find two copies: one properly filed in the history folder and one misfiled in the root. Delete the misfiled copy, then read the correctly filed one. If it mentions Argentina, add the tag Latin_America to it.	4

profile and entity scope. The generation prompt additionally controls information availability (d_{info}) by instructing the model whether to expose identifiers explicitly or require the agent to discover them via API queries. In some cases, we supply a pre-authored seed template and ask the model to generate only the prompt and assertions; in others, the model constructs all three components jointly. The full generation prompt is provided in Appendix A.11.

Human curation. Each generated task is validated for executability in the sandbox. Reviewers verify that required identifiers are discoverable and that the necessary endpoints and permissions exist under the stated prompt constraints. Reviewers then control the ambiguity dimension (d_{amb}) by selectively degrading prompts – removing explicit identifiers, introducing typographical variations, or adding distractor entities to the seed state. All tasks are annotated along the five taxonomy dimensions post-hoc to verify broad coverage (Table 2).

Benchmark summary. The resulting benchmark comprises 224 tasks across four enterprise services: Box (file management), Linear (project management), Slack (messaging), and Google Calendar (scheduling). Table 2 summarizes the distribution across taxonomy dimensions. Three design choices are worth noting: (1) implicit information dominates (66% of tasks), forcing agents to discover identifiers via API queries rather than receiving them in the prompt; (2) multi-entity tasks (53%) require coordinated state changes across multiple tables; and (3) each service exercises distinct API semantics – Box emphasizes nested hierarchies and permissions, Slack stresses threading and multi-party DMs, Calendar demands temporal reasoning and free/busy queries, and Linear involves issue dependencies and workflow state machines.

Table 2: Benchmark task distribution by service and taxonomy dimension. *Tasks*: number of evaluation scenarios per API service. n^*: task horizon, i.e. the minimum number of API endpoint invocations under an optimal policy. Operation profiles are percentages and are non-exclusive. Entity scope, information availability, and prompt ambiguity are mutually exclusive per-task categories and are reported as counts.

Metric	Box	Slack	Linear	Cal.	Total
Tasks	48	59	57	60	224
n^* (range)	1–13	1–14	1–13	1–24	1–24
n^* (mean)	4.6	5.6	5.2	5.9	5.3
Operation profile (% of tasks; non-exclusive)					
Search	92	64	89	77	80
Create	58	88	63	78	73
Read	54	68	14	82	55
Update	62	37	70	93	66
Delete	19	24	7	53	26
Entity scope (count; exclusive)					
Single-entity	28	33	33	11	105
Multi-entity	20	26	24	49	119
Information availability (count; exclusive)					
Explicit	6	36	25	10	77
Implicit	42	23	32	50	147
Prompt ambiguity (count; exclusive)					
Low	24	27	37	13	101
Medium	17	22	19	45	103
High	7	10	1	2	20

4 Evaluation Methodology

State-diff evaluation. Each task τ executes in an isolated sandbox with initial state S_0 and final state S_1. We compute a canonical state diff $\Delta S(\tau) = \text{Diff}(S_0, S_1)$, consisting of inserted, deleted, and updated entities across all tables (Appendix A.10). All verification is performed on $\Delta S(\tau)$ rather than on the agent's action trace.

Declarative verification logic. We express task requirements as a set of predicates over $\Delta S(\tau)$, avoiding brittle operation matching. Each assertion specifies (i) a `diff_type` (added, deleted, or updated), (ii) an `entity` (table), (iii) field-level constraints in `where` (e.g., equality or substring match), and (iv) an `expected_count`. This yields a compact, implementation-agnostic verification layer that can be authored without imperative database queries.

Closed-world invariant. Let $A(\tau)$ denote the required assertions for task τ, and let $F(\tau)$ denote explicitly ignored fields (e.g., non-deterministic timestamps such as `updated_at`). We enforce a closed-world invariant: every change in $\Delta S(\tau)$ must either be explained by an assertion in $A(\tau)$

Model	Per-service score				Aggregate			Efficiency		
	Box	Calendar	Linear	Slack	Overall	Pass (%)	Cost ($)	Tokens	Score/$	Rank
`deepseek-v3.2`	76.6±10.1	87.5±3.9	94.8±3.7	86.1±8.5	**88.1**±2.4	76	0.03	187k	2,938	1
`devstral-2512`	79.0±9.7	80.0±5.8	91.5±4.7	85.7±5.5	**86.0**±3.1	74	0.08	199k	1,075	2
`qwen3-vl-235b`	68.4±12.2	71.0±6.6	82.0±7.5	75.8±8.7	**79.2**±3.3	65	0.02	128k	3,959	3
`kimi-k2-0905`	66.5±8.6	72.3±6.9	88.2±5.7	82.2±7.0	**75.4**±3.5	64	0.04	144k	1,885	4
`grok-4.1-fast`	58.5±8.5	75.7±5.3	66.0±7.8	77.1±5.7	**74.9**±2.7	52	0.01	131k	7,489	5
`gemini-3-flash`	80.3±8.2	62.2±9.3	84.0±6.9	77.5±8.1	**73.8**±3.7	67	0.05	176k	1,477	6
`gpt-oss-120b`	70.1±8.9	68.4±8.2	79.5±7.4	69.1±8.9	**68.5**±4.1	60	0.02	157k	3,428	7
`claude-haiku-4.5`	45.1±15.6	57.8±10.6	35.6±15.0	57.3±13.1	**49.3**±5.5	50	0.22	172k	224	8
`llama-4-scout`	33.7±8.9	41.4±9.2	20.9±7.7	42.9±8.6	**38.0**±4.1	29	0.02	183k	1,900	9

Table 3: Model performance on Agent-Diff in the no-docs setting. Entries are assertion-weighted scores with 95% Bayesian credible intervals. Models are sorted by overall score. Best and worst values in each column are highlighted.

or involve only fields in $F(\tau)$. Any other insertion, deletion, or mutation is treated as a side effect and causes the task to fail.

Scoring and uncertainty. Let $\mathrm{sat}(a, \Delta S) \in \{0, 1\}$ indicate whether assertion a holds on the state diff. We report two task-level metrics. Pass rate is a binary indicator that equals 1 iff a task is clean and all assertions are satisfied: $\mathrm{Pass}(\tau) = \mathrm{clean}(\tau) \cdot \mathbb{1}[\forall a \in A(\tau),\ \mathrm{sat}(a, \Delta S) = 1]$. Score is an assertion-weighted metric: each task contributes $s(\tau) = \mathrm{clean}(\tau) \sum_{a \in A(\tau)} \mathrm{sat}(a, \Delta S)$ out of a maximum $m(\tau) = |A(\tau)|$, so that a task with unexpected side effects contributes zero regardless of partial progress. The aggregate score over a task set $\mathcal{T}$ is

$$\mathrm{Score}(\mathcal{T}) = \frac{\sum_{\tau \in \mathcal{T}} s(\tau)}{\sum_{\tau \in \mathcal{T}} m(\tau)}. \tag{11}$$

We quantify uncertainty with a task-level Bayesian bootstrap (Rubin, 1981). We report posterior means and 95% credible intervals over $B = 10{,}000$ draws. For documentation ablations, we use a paired Bayesian bootstrap over the shared task set and report $\Pr[\Delta > 0]$ as the posterior probability that documentation improves performance. Full details are given in Appendices A.1 and A.2.

5 Experiments

5.1 Experiment Setup

Prompting & Interaction. We utilize the ReAct prompting framework (Yao et al., 2023b) adapted for code execution. The system prompt (Appendix A.8) instructs the model to reason about the task and emit a code block (Bash) to interact with the environment.

Documentation conditions. To disentangle reasoning ability from prior API knowledge, we evaluate each model under three conditions. **No-docs** (∼400 tokens): the prompt contains only the service name, base URL, and ReAct instructions; the agent must discover endpoints by probing the API. **Relevant-docs** (∼3,200–10,000 tokens): the target service's API specification is injected into the prompt (Table 9). **All-docs** (∼22,300 tokens): specifications for all four services, testing whether irrelevant documentation degrades performance via context dilution.

Sandbox Execution Environment. Model responses are parsed to extract Bash code blocks, which are executed in a containerized environment equipped with standard Linux utilities (`jq`, `curl`, `grep`, `sed`) and a test-local filesystem for intermediate file manipulation. All network traffic is intercepted and routed to local service replicas (Slack, Linear, Box, Google Calendar), which expose the same APIs and error schemas as the upstream services.

Inference Parameters. We evaluate the models listed in Table 10 (Appendix) via the OpenRouter API using each provider's default temperature. Each model uses its full context window; episodes that exceed the context or token limit are marked as failed. An episode terminates when the agent emits a <done> token, the trace reaches 40 turns, or wall-clock time exceeds 8 minutes.

Evaluation Protocol. Each run starts from a fresh, deterministically seeded sandbox state. We perform 3 trials per (task, documentation-condition) combination across 3 conditions (no-docs, relevant-docs, all-docs), yielding 9 runs per task per model and 2,016 traces per model (224

tasks $\times$ 9). We log full conversation traces, token usage, cost, and number of tool invocations for efficiency analysis.

6 Experiment Results

6.1 Model-to-model comparison

Table 3 reports per-service and overall assertion-weighted scores under the no-docs baseline, where agents receive no API documentation and must discover endpoints through exploration.

Overall ranking. DeepSeek-v3.2 leads with an overall score of 88.1 (95% CrI: $\pm$2.4), followed closely by Devstral-2512 at 86.0. These two models form a clear top tier, separated from the next cluster (Qwen3-VL through Gemini-3-Flash, 73.8–79.2) by a gap of $\sim$7 points. The bottom tier – Claude-Haiku-4.5 (49.3) and Llama-4-Scout (38.0) – scores below 50%.

Service-level variation. No single service is universally easiest or hardest. Linear yields the highest scores for most models (grand mean 71.4), likely because its issue-tracking API follows predictable CRUD conventions. Calendar proves challenging for models that otherwise perform well: Gemini-3-Flash scores 80.3 on Box but only 62.2 on Calendar, suggesting that temporal reasoning and free/busy queries pose distinct difficulties.

Cost efficiency. Grok-4.1-Fast achieves the best cost efficiency at \$0.01/test (Score/\$ = 7,489), outperforming models 5–8$\times$ more expensive. Claude-Haiku-4.5 is both the most expensive (\$0.22/test) and among the worst performers, yielding only 224 Score/\$ – a 33$\times$ gap relative to Grok. Token consumption varies modestly (128k–199k), suggesting that cost differences are driven primarily by per-token pricing rather than verbosity.

6.2 Documentation effects

Detailed per-service and per-model results are reported in Appendix A.2. Providing relevant API documentation increases overall pass rate relative to the no-docs baseline, with the strongest gains on Box, Calendar, and Linear. Under relevant-docs, pass rate improves by $+7.0\pm2.3$ pp ($P = 1.00$), and 6 of 9 models show individually significant improvements ($P > 0.975$). However, the assertion-weighted score shows only a smaller, non-significant aggregate gain ($\bar{\Delta} = +2.4$, $P = 0.73$). This divergence suggests that documentation helps agents complete simpler tasks outright, but offers diminishing returns on complex multi-assertion tasks. Effects also remain heterogeneous across models: on Calendar, for example, relevant-docs improves Qwen3-VL by $+12.3$ points ($P = 1.00$) but hurts Claude-Haiku by -11.9 ($P = 0.02$).

By contrast, all-docs yields no reliable overall score improvement ($\bar{\Delta} = +0.2$, $P = 0.53$), despite still increasing pass rate by $+6.8$ pp ($P = 1.00$). The clearest degradation appears on Linear, where the pooled score delta is -4.0, with five models showing confident or suggestive harm. Because relevant-docs on Linear is near-neutral ($+1.7$), this pattern suggests that irrelevant documentation can compete for attention in the context window rather than help task completion.

Box Hub tasks as a novelty test. Box is the only service with a suggestive aggregate score benefit from documentation (relevant-docs: $\bar{\Delta} = +10.0$, $P = 0.92$; all-docs: $+7.6$, $P = 0.81$). This effect appears to be driven by Box Hub tasks. In July 2025, Box introduced the Hub API (Box, Inc., 2025), a set of endpoints for organizing content into shared hubs that post-date the training cutoffs of most evaluated models. This creates a natural experiment: Hub tasks test the use of genuinely novel APIs, whereas standard file and folder tasks mostly involve interfaces likely represented in pre-training. As shown in Table 4, the documentation benefit is concentrated in Hub tasks: under relevant-docs, Hub tasks improve by $\Delta_{\text{hub}} = +19.0\pm10.5$ ($P = 1.00$), compared to $\Delta_{\text{non-hub}} = +3.4\pm4.4$ ($P = 0.93$), with $P(\Delta_{\text{hub}} > \Delta_{\text{non-hub}}) = 0.999$. The contrast is especially strong for top-tier models: DeepSeek-v3.2 and Devstral-2512 gain roughly $+31$ points on Hub tasks but remain essentially flat on Non-Hub tasks. This asymmetry suggests that documentation is most useful when it provides genuinely new API knowledge rather than reiterating familiar interfaces. Figure 4 in Appendix A.2 illustrates this pattern qualitatively: without documentation, Claude-Haiku-4.5 hallucinates a Collections-based solution, whereas with relevant Box documentation it uses the correct Hub endpoints and completes the task successfully.

6.3 Failure and Recovery Modes

Error taxonomy and annotation. We annotate 4,032 traces (9 models $\times$ 224 tasks $\times$ 2 documentation conditions) using an LLM judge (Gemini 3 Flash) with a structured schema covering 32 fine-grained error types (Appendix A.13.1). For report-

Table 4: Documentation effect on Box tasks by subset (relevant-docs vs. no-docs). Δ_{hub} and $\Delta_{\text{non-hub}}$ denote posterior mean score deltas. $P(\Delta_{\text{hub}} > \Delta_{\text{non-hub}})$ gives the posterior probability that documentation benefits Hub tasks more than non-Hub tasks.

	Hub		Non-Hub		
Model	Δ	$P(>0)$	Δ	$P(>0)$	$P(\text{H} > \text{NH})$
`deepseek-v3.2`	+31.9	1.00	−1.0	0.38	1.00
`devstral-2512`	+31.3	1.00	−1.9	0.29	1.00
`qwen3-vl-235b`	+15.5	0.97	+11.6	0.95	0.65
`kimi-k2-0905`	+11.3	0.85	−2.0	0.33	0.87
`grok-4.1-fast`	+26.7	1.00	+5.6	0.91	1.00
`gemini-3-flash`	+0.4	0.54	+7.4	0.88	0.32
`gpt-oss-120b`	+17.9	1.00	−4.7	0.19	1.00
`claude-haiku-4.5`	+11.9	0.87	+2.2	0.63	0.79
`llama-4-scout`	+14.6	0.97	+13.2	0.99	0.55
Pooled	**+19.0**	**1.00**	**+3.4**	**0.93**	**1.00**
	[+9.8, +30.8]		[−0.9, +7.8]		

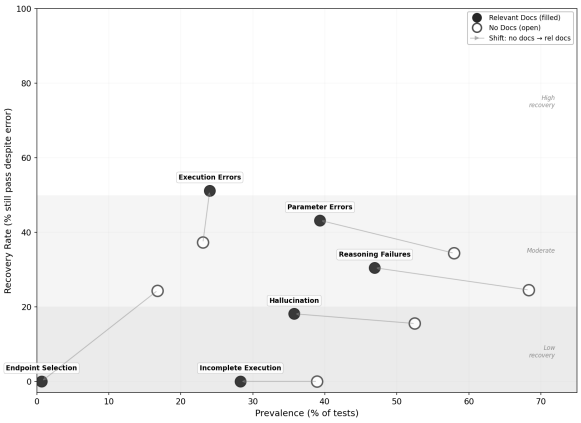

Figure 2: Error prevalence vs. recovery rate under no-docs (open circles) and relevant-docs (filled circles) conditions. Arrows show the shift when documentation is provided.

ing, these are merged into five error causes – Endpoint Selection, Parameter Errors, Execution Errors, Hallucination, and Reasoning Failures – plus one failure mode (Incomplete Execution).

Error prevalence and recoverability. Figure 2 plots each error category by prevalence (x-axis) and recovery rate (y-axis) under both documentation conditions. Three patterns emerge. First, documentation uniformly shifts categories leftward (lower prevalence), with the largest reductions for Reasoning Failures (−25.9 pp) and Hallucination (−21.2 pp). Second, recovery rates improve across all error causes when documentation is available, with Parameter Errors gaining +21.3 pp and Reasoning Failures +20.8 pp. Third, Incomplete Execution remains anchored in the low-recovery zone regardless of documentation (0.0% in both conditions), indicating that agents which fail to attempt

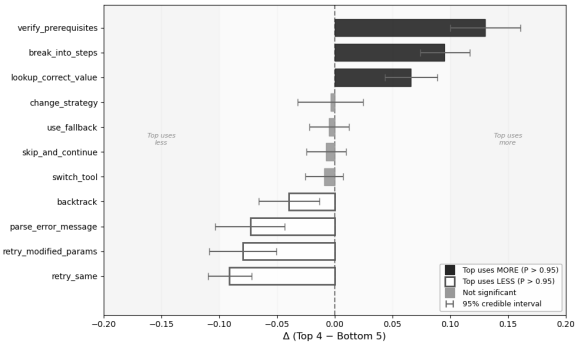

Figure 3: Analysis of recovery strategy effectiveness across model performance tiers. $\Delta = \bar{x}_{\text{top}} - \bar{x}_{\text{bottom}}$ is the posterior mean usage rate difference between top-performing models and bottom-performing models. Filled bars indicate strategies used significantly more by top models; open bars indicate strategies used significantly less ($P > 0.95$).

sub-tasks rarely recover even with better information.

Endpoint Selection is rare under relevant-docs (0.5%) but nearly always fatal: no run with this error passed in either condition. In contrast, Execution Errors are moderately prevalent (23–30%) but the most recoverable category, reaching 54.8% recovery with documentation – suggesting that agents can parse error messages and retry successfully when the endpoint choice is correct. Appendix A.4 further shows that relevant documentation reduces API-knowledge-related errors, both in absolute terms and as a share of the overall error budget.

Recovery strategies and model tiers. The most common recovery strategies are *break_into_steps* (86.8% of runs) and *lookup_correct_value* (79.8%); the least common are *wait_and_retry* (0.3%), *use_fallback* (7.4%), and *switch_tool* (7.6%). Bayesian bootstrap with task-level clustering reveals that *lookup_correct_value* (8/9 models significant) and *break_into_steps* (6/9) are the strongest positive predictors of task score, while *retry_same* (8/9) and *use_fallback* (7/9) predict failure. Comparing top-tier models (DeepSeek, Devstral, Qwen, Kimi) against bottom-tier via median split (Figure 3), top models employ *verify_prerequisites* and *break_into_steps* at significantly higher rates ($P = 1.00$), while bottom models rely on *retry_same* ($P < 0.01$). Overall, higher-performing models favor adaptive, information-seeking recovery over repetitive strategies (Appendix A.5).

7 Conclusion

We introduced Agent-Diff, a benchmark for code-executing LLM agents on enterprise API tasks. Containerized replicas of production APIs provide ecological fidelity with full reproducibility, and a state-diff evaluation contract enables reproducible, declarative assertions that verify expected state changes while enforcing a closed-world invariant to detect unintended side effects. Evaluating nine models across 224 tasks, we find scores ranging from 38.0 to 88.1, with documentation effects that are service- and model-specific: pass rates improve significantly ($+7.0$ pp, $P = 1.00$) while assertion-weighted scores do not ($+2.4$, $P = 0.73$). A training-cutoff natural experiment validates our inclusion of recently introduced endpoints, which isolate inference-time learning from pre-training memorization. Error and recovery analysis reveals that top-performing models favor adaptive strategies (lookup, decomposition) over repetitive ones (retry-same), indicating that planning quality differentiates model tiers beyond API knowledge alone. Code and infrastructure are released under MIT at `https://github.com/agent-diff-bench/agent-diff`.

References

Anthropic. 2024. Model context protocol.

Anthropic. 2026a. 2026 agentic coding trends report: How coding agents are reshaping software development. Technical report, Anthropic. Accessed: 2026-01-28. Covers trends like multi-agent teams, long-running agents, and productivity gains in SDLC.

Anthropic. 2026b. claude-code (github repository).

Jayachandu Bandlamudi, Ritwik Chaudhuri, Neelamadhav Gantayat, Sambit Ghosh, Kushal Mukherjee, Prerna Agarwal, Renuka Sindhgatta, and Sameep Mehta. 2025. A Framework for Testing and Adapting REST APIs as LLM Tools. *arXiv preprint*. ArXiv:2504.15546 [cs].

Box, Inc. 2025. Announcing the Box Hubs API — July 2025. Accessed: 2026-02-09.

Box, Inc. 2026a. Box API reference. Accessed: 2026-01-25.

Box, Inc. 2026b. box-python-sdk. GitHub repository.

Cursor. 2026. Cursor cli.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, and 3 others. 2025a. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks? *Preprint*, arXiv:2509.16941.

Xiang Deng, Jeff Da, Edwin Pan, and 1 others. 2025b. SWE-Bench Pro: Can AI agents solve long-horizon software engineering tasks? *arXiv preprint*. ArXiv:2509.16941 [cs].

Xuanqi Gao, Siyi Xie, Juan Zhai, Shiqing Ma, and Chao Shen. 2025. MCP-RADAR: A Multi-Dimensional Benchmark for Evaluating Tool Use Capabilities in Large Language Models. *arXiv preprint*. ArXiv:2505.16700 [cs].

Google. 2026. gemini-cli (github repository).

Google LLC. 2026. Google calendar API v3 reference. Accessed: 2026-02-02.

Adam Jones and Conor Kelly. 2025. Code execution with MCP: Building more efficient agents. `https://www.anthropic.com/engineering/code-execution-with-mcp`. Anthropic Engineering Blog.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. 2023. API-Bank: A Comprehensive Benchmark for Tool-Augmented LLMs. *arXiv preprint*. ArXiv:2304.08244 [cs].

Linear Inc. 2026a. Linear GraphQL API public schema. Accessed: 2026-02-02.

Linear Inc. 2026b. @linear/sdk. Npm package.

Ziyang Luo, Zhiqi Shen, Wenzhuo Yang, Zirui Zhao, Prathyusha Jwalapuram, Amrita Saha, Doyen Sahoo, Silvio Savarese, Caiming Xiong, and Junnan Li. 2025. MCP-Universe: Benchmarking Large Language Models with Real-World Model Context Protocol Servers. *arXiv preprint*. ArXiv:2508.14704 [cs].

Seiji Maekawa, Jackson Hassell, Pouya Pezeshkpour, Tom Mitchell, and Estevam Hruschka. 2025. Towards Reliable Benchmarking: A Contamination Free, Controllable Evaluation Framework for Multi-step LLM Function Calling. *arXiv preprint*. ArXiv:2509.26553 [cs].

Tula Masterman, Sandi Besen, Mason Sawtell, and Alex Chao. 2024. The landscape of emerging AI agent architectures for reasoning, planning, and tool calling: A survey. *arXiv preprint*. ArXiv:2404.11584 [cs].

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Jenia Jitsev, Di Lu, Orfeas Menis Mastromichalakis, and 66 others. 2026. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *Preprint*, arXiv:2601.11868.

Donald B. Rubin. 1981. The bayesian bootstrap. *The Annals of Statistics*, 9(1):130–134.

Slack Technologies. 2026a. python-slack-sdk. GitHub repository.

Slack Technologies. 2026b. Slack web API methods. Accessed: 2026-02-02.

Zhenting Wang, Qi Chang, Hemani Patel, Shashank Biju, Cheng-En Wu, Quan Liu, Aolin Ding, Alireza Rezazadeh, Ankit Shah, Yujia Bao, and Eugene Siow. 2025. MCP-Bench: Benchmarking Tool-Using LLM Agents with Complex Real-World Tasks via MCP Servers. *arXiv preprint*. Version Number: 1.

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. 2025. MCPWorld: A Unified Benchmarking Testbed for API, GUI, and Hybrid Computer Use Agents. *arXiv preprint*. ArXiv:2506.07672 [cs].

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

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023a. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. *arXiv preprint*. ArXiv:2305.10601 [cs].

Shunyu Yao, Jeffrey Zhao, Dian Yu, and 1 others. 2023b. React: Synergizing reasoning and acting in language models. *arXiv preprint*. ArXiv:2210.03629 [cs].

A Appendices

A.1 Score Uncertainty Evaluation

We quantify uncertainty via the Bayesian bootstrap (Rubin, 1981) with a uniform Dirichlet prior, placing no *a priori* preference among tasks. For a set of n tasks, we draw task weights $\mathbf{w} \sim \text{Dirichlet}(\mathbf{1}_n)$ and compute the assertion-weighted score

$$\hat{S}^{(b)} = \frac{\sum_i w_i^{(b)} s_i}{\sum_i w_i^{(b)} m_i}, \tag{12}$$

where s_i is the mean raw score for task i (averaged over trials) and $m_i = |A(\tau_i)|$ is the maximum possible score. We repeat for $B = 10{,}000$ draws, reporting the posterior mean and 95% credible interval (CrI) from the empirical distribution of $\{\hat{S}^{(b)}\}_{b=1}^{B}$. The $\text{Dirichlet}(\mathbf{1}_n)$ prior is the standard non-informative choice, treating all tasks as equally likely before reweighting.

No-docs (passed: No)	**Relevant-docs** (passed: Yes)
Step 1: Lists root folder `GET /folders/0` ✓	*Step 1:* Creates hub directly `POST /hubs` ✓
Step 2: Lists folder contents `GET /folders/{id}/items` ✓	*Step 2:* Searches for folder `GET /search?query=...` ✓
Step 3: Attempts Collections API `PUT /files/{id}` → `null` ×	*Step 3:* Lists folder items `GET /folders/{id}/items` ✓
Step 4: Checks collections `GET /collections` → only `Favorites`	*Step 4:* Adds all files to hub `POST /hubs/{id}/manage_items` ✓
Step 5: Hallucinates success	*Step 5:* Verifies hub contents `GET /hub_items?hub_id=...` ✓

Figure 4: Claude-Haiku-4.5 on *Organize Research Hub* ($n^* = 3$). Left: without documentation, the model tries the Collections API, receives null responses, and incorrectly concludes that the task succeeded. Right: with relevant Box documentation, it uses the correct Hub endpoints (`POST /hubs`, `POST /hubs/{id}/manage_items`) and completes the task successfully. Full trace in Appendix A.14.

A.2 Documentation Effects

For the documentation ablation, we compare three conditions: *no_docs* (baseline), *relevant_docs* (target service only), and *all_docs* (all four services). We use a paired Bayesian bootstrap: the *same* Dirichlet draw $\mathbf{w}^{(b)}$ is applied to both conditions over their shared task set, yielding a posterior over $\Delta^{(b)} = \hat{S}_A^{(b)} - \hat{S}_B^{(b)}$ with correctly correlated credible intervals on the treatment effect. We report $\Pr[\Delta > 0]$ as the posterior probability that documentation improves performance. Cells with $\Pr[\Delta > 0] > 0.9$ indicate confident beneficial effects; $\Pr[\Delta > 0] < 0.1$ indicates confident harmful effects. Table 5 shows the results of the experiments.

A.3 Error Prevalence Analysis

We analyze error prevalence – the frequency with which different error types manifest across model runs. This complements the recovery strategy analysis by characterizing *what goes wrong* rather than *how agents respond*. Understanding error patterns helps identify systematic weaknesses in agent capabilities.

A.3.1 Error Category Taxonomy

Through qualitative coding of agent trajectories, we identified errors across three high-level domains

Model	Box		Calendar		Linear		Slack		Overall	
	Relevant	All docs	Relevant	All docs	Relevant	All docs	Relevant	All docs	Relevant	All docs
deepseek-v3.2	+14.1 (1.00)	+16.2 (1.00)	−0.5 (0.39)	+0.1 (0.58)	+0.9 (0.87)	−9.2 (0.00)	−1.5 (0.36)	+4.3 (0.94)	+1.4 (0.87)	+1.1 (0.75)
devstral-2512	+13.4 (1.00)	+8.0 (0.97)	+3.0 (0.88)	+4.2 (0.99)	−1.4 (0.28)	−3.9 (0.02)	+1.0 (0.65)	+3.6 (0.96)	+2.8 (0.96)	+2.7 (0.99)
qwen3-vl-235b	+13.6 (1.00)	+12.1 (0.95)	+12.3 (1.00)	+13.5 (1.00)	+6.5 (0.98)	−2.2 (0.37)	−0.2 (0.49)	+3.3 (0.89)	+7.9 (1.00)	+7.1 (1.00)
kimi-k2-0905	+4.0 (0.77)	+13.6 (1.00)	−4.1 (0.20)	−4.0 (0.17)	−7.7 (0.05)	−13.4 (0.00)	+1.1 (0.67)	−2.0 (0.26)	−2.6 (0.14)	−3.4 (0.06)
grok-4.1-fast	+15.2 (1.00)	+15.0 (1.00)	+3.1 (0.93)	+0.8 (0.64)	+12.8 (0.99)	+9.2 (0.99)	−5.8 (0.06)	+2.8 (0.80)	+4.6 (0.99)	+5.1 (1.00)
gemini-3-flash	+4.1 (0.73)	+1.3 (0.62)	+4.1 (0.83)	+3.7 (0.76)	−0.3 (0.48)	−10.7 (0.03)	−1.2 (0.39)	+2.1 (0.70)	+1.6 (0.74)	−0.5 (0.42)
gpt-oss-120b	+5.5 (0.88)	+1.8 (0.61)	−6.9 (0.01)	−4.3 (0.08)	+2.1 (0.72)	−14.9 (0.01)	+1.6 (0.64)	−0.3 (0.45)	−0.9 (0.32)	−5.0 (0.01)
claude-haiku-4.5	+6.6 (0.86)	−4.5 (0.28)	−11.9 (0.02)	−9.1 (0.06)	+2.5 (0.64)	+4.1 (0.77)	+14.2 (0.96)	−7.0 (0.21)	+0.4 (0.55)	−5.0 (0.09)
llama-4-scout	+13.8 (1.00)	+5.0 (0.84)	+4.0 (0.78)	−2.8 (0.21)	−0.2 (0.50)	+4.8 (0.90)	+11.3 (1.00)	−4.2 (0.18)	+6.1 (0.98)	−0.4 (0.43)
Average score	**+10.0 (0.92)**	**+7.6 (0.81)**	**+0.3 (0.56)**	**+0.2 (0.50)**	**+1.7 (0.61)**	**−4.0 (0.34)**	**+2.3 (0.58)**	**+0.3 (0.60)**	**+2.4 (0.73)**	**+0.2 (0.53)**
Pass-rate delta (pp)										
Average pass	**+14.0 (1.00)**	**+14.8 (1.00)**	**+8.4 (1.00)**	**+7.6 (1.00)**	**+6.3 (1.00)**	**+4.1 (1.00)**	**+0.5 (0.61)**	**+2.4 (0.93)**	**+7.0 (1.00)**	**+6.8 (1.00)**

Table 5: Per-service documentation effects, $\Delta = \bar{S}_{\text{docs}} - \bar{S}_{\text{no_docs}}$. Per-model entries report assertion-weighted score deltas with posterior probability $P(\Delta > 0)$ from a paired Bayesian bootstrap ($B = 10{,}000$). Bottom rows show pooled score and pass-rate deltas averaged across models. Cell colors indicate beneficial or harmful effects by posterior confidence.

(tool use, hallucination, and reasoning), which we consolidated into 6 merged categories for interpretability: endpoint selection, parameter errors, execution errors, reasoning failures, and incomplete execution. Each error category is coded as binary (present/absent) per run, allowing a run to exhibit multiple error types simultaneously. More information on each of those categories is presented in Appendix A.13.1.

A.3.2 Prevalence Rate Analysis

Table 6 reports the percentage of runs in which each error category was observed, stratified by model. This analysis covers all $n = 3{,}955$ analyzed runs.

Computation. For each model m and error category e, we compute:

$$\text{Prevalence}_{m,e} = \frac{|\{r \in R_m : e \in r\}|}{|R_m|} \times 100 \quad (13)$$

where R_m denotes the set of runs for model m, and $e \in r$ indicates that error category e was coded as present in run r.

Category Merging. The 6 merged categories consolidate 35 fine-grained error annotations as follows:

- *endpoint_selection* ← `tool_use_errors.endpoint_selection`
- *parameter_errors* ← `tool_use_errors.parameter_misuse` ∨ `hallucination_errors.parameter_hallucination`
- *execution_errors* ← `tool_use_errors.format_errors` ∨ `tool_use_errors.code_errors`
- *hallucination* ← ⋁ {`outcome`, `state`, `action`, `context`, `capability`, `other`}-hallucination
- *reasoning_failures* ← ⋁ {*16 reasoning error subtypes, excluding incomplete/premature*}
- *incomplete_execution* ← `reasoning_errors.incomplete_execution_error` ∨ `reasoning_errors.premature_termination_error`

Interpretation. The heatmap reveals that *reasoning_failures* (53.6% overall) and *parameter_errors* (43.3%) are the most prevalent error categories, suggesting that logical reasoning and correct parameterization remain challenging for current agents. *Hallucination* affects over one-third of runs (36.8%), indicating persistent grounding issues.

endpoint_selection errors are relatively rare (10.3%), suggesting that models generally identify the correct API operations even when they struggle with parameter details or execution.

Notably, `llama-4-scout` exhibits substantially elevated error rates across all categories – particularly *reasoning_failures* (86.5% vs. 53.6% overall) and *parameter_errors* (80.9% vs. 43.3% overall). This model also shows the highest *endpoint_selection* error rate (35.0% vs. 10.3% overall), a 3.4× elevation that is unique among evaluated models.

In contrast, `deepseek-v3.2` demonstrates the lowest error rates across most categories, with particularly low *hallucination* (12.3%) and *incomplete_execution* (8.0%) rates, consistent with its position as a top-performing model.

Model	Endpoint	Execution	Halluc.	Incomplete	Parameter	Reasoning
`claude-haiku-4.5`	5.4	18.6	48.4	44.3	39.6	61.8
`deepseek-v3.2`	5.7	10.2	12.3	8.0	24.1	32.0
`devstral-2512`	9.5	18.9	20.0	13.7	34.7	42.1
`gemini-3-flash-preview`	2.7	9.9	34.1	23.0	34.1	41.1
`gpt-oss-120b`	6.8	25.8	45.9	30.5	49.1	60.6
`grok-4.1-fast`	9.3	20.2	48.6	38.9	50.9	63.0
`kimi-k2-0905`	9.3	12.8	30.8	22.6	34.4	44.6
`llama-4-scout`	35.0	45.4	61.7	64.3	80.9	86.5
`qwen3-vl-235b-a22b-instruct`	9.7	15.4	30.8	21.7	43.7	52.3
Overall	**10.3**	**19.6**	**36.8**	**29.5**	**43.3**	**53.6**

Table 6: Error prevalence rates by model (% of runs where each error type was observed). Darker shading indicates higher error rates. Based on 3955 analyzed runs across 6 merged error categories.

A.4 Bayesian Bootstrap Analysis of API Knowledge Errors

To assess whether documentation reduces API knowledge errors, we use a Bayesian bootstrap (Rubin, 1981) that accounts for test-level dependence and yields posterior uncertainty estimates.

Error definition. We define API knowledge errors as the sum of two error categories extracted from our qualitative coding of agent trajectories:

- **Endpoint selection errors**: calls to non-existent endpoints, use of deprecated APIs, or selection of inappropriate endpoints for the task.
- **Parameter misuse**: incorrect argument types, missing required fields, malformed values, or violations of parameter constraints.

These categories capture failures attributable to incomplete or incorrect knowledge of the API specification, as distinct from reasoning errors, execution failures, or environmental issues.

Estimand. For each model, we estimate

$$\Delta = \mathbb{E}[\text{API knowledge errors} \mid \texttt{no_docs}] - \mathbb{E}[\text{API knowledge errors} \mid \texttt{with_docs}]$$

so that positive values indicate fewer API knowledge errors when documentation is provided.

For each model, we compare runs under two conditions: `no_docs` (no API documentation provided) and `with_docs` (relevant API documentation included in context).

Clustering by test. Because the same test may appear under both documentation conditions, observations are not independent at the run level. Therefore, we restrict analysis to tests observed in both conditions and treat `test_id` as the resampling unit. For each shared test, we compute a condition-specific test-level summary, then compare these summaries across conditions. Bayesian bootstrap reweighting is applied at the test level rather than the run level, analogously to a paired bootstrap while allowing unequal numbers of runs per test.

Bootstrap procedure. For each of $n = 100{,}000$ draws:

1. Sample Dirichlet weights $\mathbf{w} \sim \text{Dir}(\mathbf{1})$ over the set of shared tests.
2. Compute the weighted test-level statistic for each condition.
3. Record the delta $\Delta = \text{statistic}(\texttt{no_docs}) - \text{statistic}(\texttt{with_docs})$.

From the resulting posterior distribution of deltas, we extract:

- Posterior mean $\bar{\Delta}$
- 95% credible interval $[\Delta_{2.5}, \Delta_{97.5}]$
- Posterior probability $P(\Delta > 0)$ and $P(\Delta < 0)$

Reproducibility. All analyses use a fixed random seed (42).

Two complementary metrics. We report two distinct statistics:

1. **Ratio** (Figure 5): API knowledge errors as a proportion of total errors, computed as

$$\text{ratio} = \frac{\sum_i w_i \cdot \text{api_errors}_i}{\sum_i w_i \cdot \text{total_errors}_i} \quad (14)$$

This metric answers: What fraction of errors are API-knowledge-related?

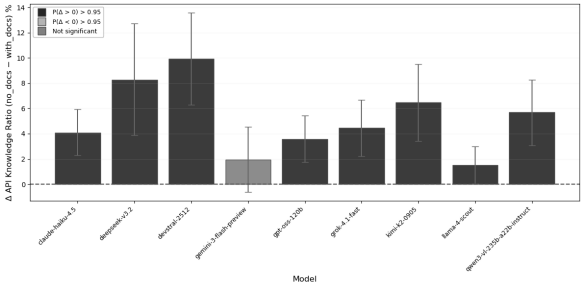

Figure 5: Effect of documentation on API knowledge error proportion (endpoint selection + parameter misuse as fraction of total errors). $\Delta > 0$ indicates documentation reduces the API error share. Shading reflects significance: dark ($P(\Delta > 0) > 0.95$), light ($P(\Delta < 0) > 0.95$), medium (not significant). Bayesian bootstrap with test-level clustering; 95% credible intervals shown.

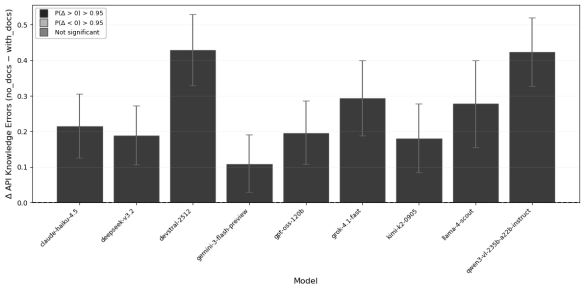

Figure 6: Effect of documentation on API knowledge error count (endpoint selection + parameter misuse per test). $\Delta > 0$ indicates documentation reduces raw error frequency. Shading reflects significance: dark ($P(\Delta > 0) > 0.95$), light ($P(\Delta < 0) > 0.95$), medium (not significant). Bayesian bootstrap with test-level clustering; 95% credible intervals shown.

2. **Absolute rate** (Figure 6): API knowledge errors per test, computed as the weighted mean

$$\text{rate} = \sum_i w_i \cdot \text{api_errors}_i \qquad (15)$$

This metric answers: How many API knowledge errors occur per test?

The distinction matters because a reduction in the ratio could arise from either (a) fewer API knowledge errors, or (b) more non-API errors inflating the denominator. By reporting both metrics, we can distinguish these scenarios:

- If ratio decreases but absolute rate is unchanged: documentation increases other error types, not a genuine improvement.
- If absolute rate decreases but ratio is unchanged: documentation reduces all error types proportionally.
- If both decrease: documentation targets API knowledge gaps more than non-API knowledge gaps.

Reproducibility. The bootstrap analysis uses a fixed random seed (42) for reproducibility.

A.5 Recovery Strategies Analysis

We analyze recovery strategies – the behavioral patterns agents exhibit when encountering errors or obstacles during task execution. This analysis proceeds in two stages: first, we characterize the frequency with which different strategies are employed across models (usage rates); second, we evaluate whether employing each strategy is associated with better or worse task outcomes (scores).

A.5.1 Recovery Strategy Taxonomy

Through qualitative coding of agent trajectories, we identified 13 distinct recovery strategies (plus an "other" category), descriptions of which are available in Appendix A.13.2. Each strategy is coded as binary (present/absent) per run, allowing a run to exhibit multiple strategies.

A.5.2 Usage Rate Analysis

Table 7 reports the percentage of runs in which each recovery strategy was observed, stratified by model. We restrict this analysis to the $n = 3{,}625$ runs where at least one recovery strategy was identified (i.e., excluding runs with no observable recovery behavior; note, that no change of recovery strategy when needed is expected to be handled by the *retry_same* case). The percentage was calculated following the same procedure as the prevalence of errors in Appendix A.3 but now applied to recovery strategies.

Interpretation. The heatmap reveals substantial cross-model consistency in strategy preferences: *break_into_steps* (86.8% overall) and *lookup_correct_value* (79.8%) dominate across all models, suggesting these are near-universal recovery behaviors for capable agents. Conversely, *retry_same* (12.1%) and *wait_and_retry* (0.3%) are rarely employed, indicating that modern agents generally avoid naive retry loops.

Notable model-specific deviations include `llama-4-scout`, which exhibits elevated usage of *retry_same* (40.8% vs. 12.1% overall) and *parse_error_message* (73.8% vs. 46.3% overall), suggesting a qualitatively different error-handling style.

Model	Back.	Steps	Strat.	Lookup	Other	Parse	Retry mod.	Retry	Skip	Switch	Fallback	Verify	Wait
`claude-haiku-4.5`	20.2	85.6	34.9	76.7	0.6	43.5	43.5	13.3	8.3	5.8	5.5	50.1	1.1
`deepseek-v3.2`	17.8	97.9	38.6	88.1	0.2	34.3	39.3	2.8	7.0	7.0	10.0	69.2	0.0
`gemini-3-flash-preview`	21.8	85.3	31.9	85.8	0.2	33.6	39.7	8.8	5.1	6.9	4.9	47.5	0.2
`llama-4-scout`	32.7	66.8	65.7	61.3	0.0	73.8	75.9	40.8	23.8	15.4	11.5	20.2	0.3
`devstral-2512`	16.4	91.8	49.1	82.2	0.9	48.6	53.0	7.0	11.0	6.3	6.8	54.9	0.0
`kimi-k2-0905`	18.7	92.4	34.9	83.5	0.5	36.6	43.0	6.9	11.1	5.4	6.6	48.4	0.2
`gpt-oss-120b`	14.4	88.0	36.0	79.4	0.0	44.9	56.9	18.3	6.8	4.2	8.1	40.7	0.8
`qwen3-vl-235b-a22b-instruct`	17.8	85.6	39.9	79.3	0.2	48.6	47.1	11.5	13.9	9.6	4.8	39.4	0.5
`grok-4.1-fast`	21.4	86.2	36.7	79.4	0.0	54.1	51.9	2.7	13.6	8.3	8.3	43.2	0.0
Overall	**20.0**	**86.8**	**40.8**	**79.8**	**0.3**	**46.3**	**49.8**	**12.1**	**11.1**	**7.6**	**7.4**	**46.3**	**0.3**

Table 7: Recovery strategy usage rates by model (% of runs where each strategy was attempted). Darker shading indicates higher usage. Based on 3625 runs with at least one recovery strategy.

A.6 Per-Model Strategy Effectiveness

Usage rates alone do not establish whether a strategy helps or harms task performance. A frequently used strategy may be common either because it is effective or because agents repeatedly attempt it despite limited benefit. To separate usage frequency from performance association, we estimate strategy-specific score differences within each model using the Bayesian bootstrap (Rubin, 1981).

Estimand. For each model m and strategy s, we estimate

$$\Delta_{m,s} = \mathbb{E}[\text{Score} \mid s = 1, m] - \mathbb{E}[\text{Score} \mid s = 0, m],$$

where $s = 1$ indicates that the strategy was used in a run and $s = 0$ indicates that it was not. Positive values of $\Delta_{m,s}$ indicate that, within model m, runs using strategy s are associated with higher scores on average.

Clustering by test. Multiple runs may correspond to the same underlying test, inducing dependence among observations. To account for this, we treat `test_id` as the resampling unit and perform Bayesian bootstrap reweighting at the test level rather than the run level.

Bootstrap procedure. For each of B draws:

1. Sample Dirichlet weights $\mathbf{w} \sim \text{Dir}(\mathbf{1})$ over the set of test clusters independently for the strategy-used and strategy-not-used groups.
2. Compute the weighted mean score for each group.
3. Record the delta $\Delta = \bar{S}_{\text{used}} - \bar{S}_{\text{not-used}}$.

From the resulting posterior distribution of deltas, we extract:

- Posterior mean $\bar{\Delta}$
- 95% credible interval $[\Delta_{2.5}, \Delta_{97.5}]$
- Posterior probability $P(\Delta > 0)$

Minimum usage threshold. We exclude strategies with usage below 5% within a model to avoid unstable estimates driven by extremely sparse cells.

Reproducibility. All bootstrap analyses use a fixed random seed (42).

A.6.1 Interpreting Effectiveness Results

Table 8 presents the per-model effectiveness rankings. Several patterns emerge:

Universally beneficial strategies. *lookup_correct_value* and *break_into_steps* rank in the top 2–3 positions for nearly all models, with consistently positive deltas ($\Delta = +6$ to $+31$ points) and high confidence ($P > 0.95$ or $P > 0.90$). These strategies likely succeed because they address fundamental sources of API errors: incorrect parameter values and task complexity.

Universally harmful strategies. *retry_same* appears in the bottom 1–2 positions for all models, with strongly negative deltas ($\Delta = -10$ to -36 points, $P > 0.95$). This confirms intuition: repeating failed actions without modification is counterproductive. Notably, the magnitude of harm is larger for higher-performing models, possibly because these models encounter *retry_same* situations only when genuinely stuck.

Model-dependent strategies. *parse_error_message* shows heterogeneous effects: beneficial for `gpt-oss` ($\Delta = +3.8$) but harmful for `deepseek-v3` ($\Delta = -5.5$, $P = 0.98$) and `grok-4.1` ($\Delta = -9.1$, $P = 0.99$). This may reflect differences in how models utilize error information – some may overfit to misleading error messages while others extract actionable guidance.

Model	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11
claude-haiku	break_steps +31.5 (1.00)	lookup_val +17.3 (1.00)	verify_prereq +13.0 (1.00)	use_fallback −0.4 (0.49)	retry_mod −3.5 (0.76)	change_strat −7.9 (0.94)	switch_tool −9.8 (0.86)	skip_cont −12.3 (0.96)	retry_same −13.7 (0.98)	parse_err −13.7 (1.00)	backtrack −16.9 (1.00)
deepseek-v3	lookup_val +6.7 (0.93)	verify_prereq +3.4 (0.88)	switch_tool −5.5 (0.84)	parse_err −5.5 (0.98)	retry_mod −6.6 (0.99)	use_fallback −7.2 (0.94)	change_strat −7.5 (1.00)	backtrack −7.9 (0.99)	break_steps −8.2 (1.00)	skip_cont −11.6 (0.98)	retry_same −14.2 (0.97)
devstral	lookup_val +13.4 (1.00)	verify_prereq +1.9 (0.76)	switch_tool +1.3 (0.66)	break_steps +0.7 (0.53)	backtrack −1.2 (0.59)	parse_err −3.8 (0.92)	retry_mod −4.0 (0.93)	change_strat −4.9 (0.97)	retry_same −10.9 (0.97)	skip_cont −11.4 (1.00)	use_fallback −17.1 (1.00)
gemini-flash	lookup_val +23.3 (1.00)	break_steps +15.7 (1.00)	verify_prereq +8.5 (0.99)	change_strat +0.1 (0.52)	skip_cont +0.1 (0.53)	retry_mod −0.4 (0.54)	backtrack −1.1 (0.58)	parse_err −2.0 (0.71)	switch_tool −2.6 (0.61)	use_fallback −10.5 (0.92)	retry_same −22.9 (1.00)
gpt-oss	break_steps +9.1 (0.95)	lookup_val +6.6 (0.92)	parse_err +3.8 (0.85)	verify_prereq +3.5 (0.84)	backtrack +1.3 (0.63)	retry_same +0.8 (0.59)	retry_mod +0.4 (0.55)	change_strat −0.3 (0.52)	use_fallback −2.7 (0.65)	skip_cont −7.1 (0.92)	switch_tool −14.5 (0.93)
grok-4.1	backtrack +1.1 (0.61)	lookup_val +0.5 (0.54)	verify_prereq −0.1 (0.50)	break_steps −1.2 (0.60)	change_strat −7.6 (0.97)	parse_err −9.1 (0.99)	switch_tool −10.5 (0.96)	retry_mod −10.9 (1.00)	skip_cont −17.2 (1.00)	use_fallback −18.2 (1.00)	retry_same −32.2 (0.99)
kimi-k2	lookup_val +27.6 (1.00)	break_steps +21.3 (1.00)	verify_prereq +12.8 (1.00)	retry_mod +2.6 (0.77)	backtrack +1.4 (0.64)	switch_tool −2.1 (0.60)	parse_err −2.5 (0.74)	change_strat −2.9 (0.78)	skip_cont −4.9 (0.86)	retry_same −14.9 (0.98)	use_fallback −19.9 (1.00)
llama-scout	break_steps +16.3 (1.00)	lookup_val +12.5 (1.00)	verify_prereq +3.9 (0.75)	skip_cont −5.0 (0.84)	backtrack −5.5 (0.87)	parse_err −9.6 (0.97)	change_strat −11.2 (0.99)	use_fallback −13.4 (0.97)	retry_mod −16.9 (1.00)	switch_tool −18.4 (1.00)	retry_same −30.8 (1.00)
qwen3-vl	break_steps +20.3 (1.00)	lookup_val +17.3 (1.00)	verify_prereq +4.5 (0.91)	parse_err −9.1 (1.00)	backtrack −10.2 (0.99)	retry_mod −12.0 (1.00)	switch_tool −12.0 (0.98)	change_strat −13.7 (1.00)	skip_cont −16.0 (1.00)	use_fallback −25.7 (1.00)	retry_same −36.2 (1.00)

Table 8: Per-model recovery strategy effectiveness ranked by score delta ($\Delta = \bar{S}_{\text{used}} - \bar{S}_{\text{not_used}}$). Each cell reports strategy, delta, and posterior probability from Bayesian bootstrap ($B = 10{,}000$) clustered by `test_id`. Green: beneficial ($P(\Delta > 0) \geq 0.95$). Blue: suggestive beneficial ($0.90 \leq P(\Delta > 0) < 0.95$). Orange: suggestive harmful ($0.90 \leq P(\Delta < 0) < 0.95$). Red: harmful ($P(\Delta < 0) \geq 0.95$).

Causal interpretation caveats. While we frame Δ as an effect, the observational nature of our data precludes strong causal claims. The association between strategy use and outcomes may be confounded by:

- **Task difficulty**: Harder tasks may elicit both more recovery attempts and lower scores.
- **Agent capability**: More capable agents may both employ sophisticated strategies and achieve higher scores independently.
- **Error type**: Certain error types may trigger specific strategies while also being intrinsically harder to recover from.

We mitigate these concerns through within-model comparisons (controlling for capability) and test-level clustering (partially controlling for task difficulty), but residual confounding remains possible. The effectiveness estimates should be interpreted as associations conditional on model and test, not as guaranteed performance improvements from strategy adoption.

A.7 Median Split Analysis: Recovery Strategy Usage by Model Performance

To investigate whether high-performing and low-performing models differ systematically in their recovery strategy usage, we conducted a median split analysis comparing strategy adoption rates between performance tiers.

A.7.1 Model Performance Ranking and Group Assignment

Models were ranked by their overall benchmark performance scores and divided into two groups at the median. This yielded the following assignment:

- **Top 4 Models (High Performers):** `deepseek-v3.2`, `devstral-2512`, `qwen3-vl-235b`, `kimi-k2-0905`
- **Bottom Models (Low Performers):** `grok-4.1-fast`, `gemini-3-flash`, `gpt-oss-120b`, `claude-haiku-4.5`, `llama-4-scout` (excluded in sensitivity analysis)

A.7.2 Llama Exclusion Rationale

We conducted two parallel analyses: one including all 9 models (Top 4 vs Bottom 5) and one excluding `Llama-4-Scout` (Top 4 vs Bottom 4). We conducted a test excluding Llama as a sensitivity analysis, since Llama would be classified as a statistical outlier using the Median Absolute Deviation (MAD) method. The modified Z-score for each model was computed as:

$$M_i = \frac{0.6745 \times (x_i - \tilde{x})}{\text{MAD}} \tag{16}$$

where $\tilde{x}$ is the median score (74.9) and MAD is the median absolute deviation (6.4). This procedure assumes that the model scores $\{x_i\}$ are measured on a common scale and are comparable across models. We treat the median $\tilde{x}$ as a robust estimate of the central tendency and $\mathrm{MAD} = \mathrm{median}(|x_i - \tilde{x}|)$ as an estimate of dispersion. The constant 0.6745 is included to put the statistic on an approximate standard normal z-score scale under a normal reference model: if $Z \sim \mathcal{N}(0, 1)$, then $\mathrm{median}(|Z|) = 0.6745$, implying $\sigma \approx \mathrm{MAD}/0.6745$ for normal data. Using the standard threshold of $|M| > 3.5$, `llama-4-scout` ($M = -3.89$) was identified as a statistical outlier.

Llama's classification as a statistical outlier may indicate a meaningfully different recovery-strategy profile that both (1) places it at a comparative disadvantage under our evaluation setup and (2) exerts disproportionate influence on summary statistics and group-level comparisons.

The primary analysis includes Llama, with the exclusive analysis reported for transparency.

A.7.3 Strategy Filtering

To ensure reliable estimates, we excluded recovery strategies with very low usage rates. Specifically, a strategy was included in the analysis only if it met a minimum 5% usage threshold in at least one of the two groups. Excluded strategies (typically *use_fallback*, *wait_and_retry*, *handle_ui_obstacle*) were documented but not analyzed.

A.7.4 Bayesian Bootstrap Procedure

For each recovery strategy passing the usage threshold, we computed the difference in usage rates between the top and bottom groups using Bayesian bootstrap (Rubin, 1981) with test-level clustering.

Estimand. For each strategy s, we compute:

$$\Delta_s = \hat{p}_s^{\text{top}} - \hat{p}_s^{\text{bottom}} \tag{17}$$

where $\hat{p}_s^{\text{group}}$ is the weighted proportion of runs in that group that employed strategy s. A positive Δ indicates higher usage among top-performing models; a negative Δ indicates higher usage among bottom-performing models.

Bootstrap Resampling. We used a paired clustered Bayesian bootstrap.

1. **Cluster definition:** Runs were clustered by a composite key combining `test_id` and `include_api_docs`, treating each (test case $\times$ documentation condition) pair as a separate experimental unit.
2. **Paired weighting:** For each bootstrap draw, we generated a single set of $\mathrm{Dirichlet}(1, 1, \ldots, 1)$ weights over the shared clusters. The same cluster weight was applied to both the top-performing and bottom-performing model groups to ensure that when comparing strategy usage rates, both groups are evaluated on identically weighted tasks.
3. **Within-group size correction:** Within each group, observations sharing a cluster divided that cluster's weight equally. If cluster c received Dirichlet weight w_c and contained $n_A^{(c)}$ observations in group A, each observation received weight $w_c/n_A^{(c)}$.
4. **Statistic computation:** For each draw, we computed the weighted usage rate in each group and took their difference: $\Delta = \hat{p}_{\text{top}} - \hat{p}_{\text{bottom}}$.

Formally, the procedure computes:

$$\Delta = \sum_c w_c \cdot \left(\bar{X}_A^{(c)} - \bar{X}_B^{(c)} \right) \tag{18}$$

where $w_c \sim \mathrm{Dirichlet}(1, \ldots, 1)$, and $\bar{X}_A^{(c)}$ is the mean strategy usage within cluster c for group A.

We used 10,000 bootstrap draws with random seed 42 for reproducibility.

From the resulting posterior distribution of deltas, we extract:

- Posterior mean $\mathbb{E}[\Delta_s]$
- 95% credible interval $[\Delta_{2.5}, \Delta_{97.5}]$
- Posterior probabilities $P(\Delta_s > 0)$ and $P(\Delta_s < 0)$

A.7.5 Sensitivity Analysis: Effect of Llama Exclusion

Excluding `lama-4-Scout` from the bottom group reveals several shifts in the comparative analysis, as represented in Figure 7.

Effect of Llama-4-Scout Exclusion. Robust findings. Three strategies – *verify_prerequisites*, *break_into_steps*, and *lookup_correct_value* – remain significantly more common among top-performing models regardless of Llama's inclusion. These represent proactive, forward-looking

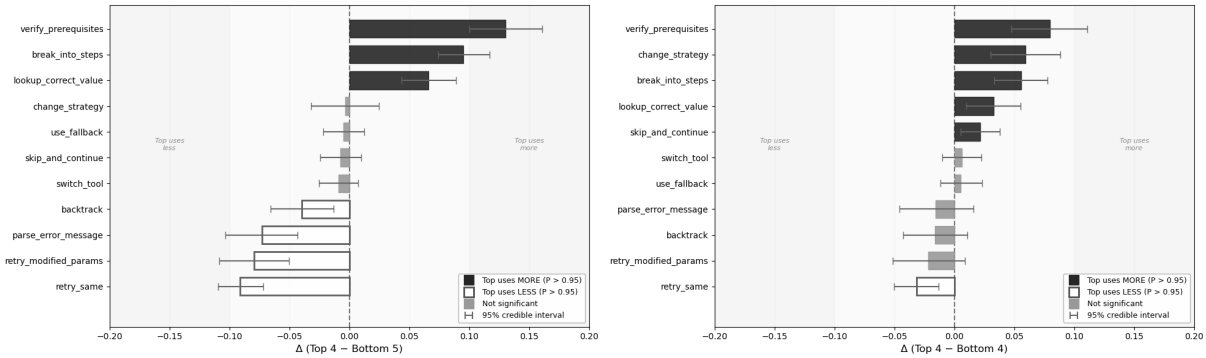

Figure 7: Recovery strategy usage differences between top-performing and bottom-performing models. Left: includes Llama (Top 4 vs Bottom 5); right: excludes Llama (Top 4 vs Bottom 4). Dark bars: significantly more common in top models ($P(\Delta > 0) > 0.95$); open bars: significantly less common ($P(\Delta < 0) > 0.95$); gray bars: not significant. Three proactive strategies (*verify_prerequisites*, *break_into_steps*, *lookup_correct_value*) and one reactive strategy (*retry_same*) show robust effects across both analyses. Excluding Llama reveals two additional strategies favoring top models (*change_strategy*, *skip_and_continue*) and attenuates effects for *retry_modified_params* and *backtrack*, suggesting Llama's outlier behavior partially masked the underlying structure.

behaviors: verifying preconditions before acting, decomposing problems into manageable steps, and retrieving authoritative information. Similarly, *retry_same* remains significantly less common among top performers in both analyses, indicating that simply repeating failed actions without modification is a reliably poor strategy.

Llama-driven effects. The *parse_error_message* strategy was excluded from the analysis when Llama was removed, indicating that it fell below the 5% usage threshold in one or both groups. This suggests Llama disproportionately relied on parsing error messages.

Masked effects. Two strategies – *change_strategy* and *skip_and_continue* – emerged as significantly more common among top performers only after Llama was excluded. This pattern suggests that Llama's usage of these strategies obscured a genuine difference between performance tiers.

Attenuated effects. The strategies *retry_modified_params* and *backtrack* shifted from significantly underrepresented among top performers to non-significant when Llama was excluded. This indicates that Llama's behavior amplified the apparent gap.

Implications. These results reveal that `llama-4-scout` exhibits qualitatively distinct recovery behavior. Specifically:

1. **Reactive over proactive:** Llama's reliance on *parse_error_message* and elevated use of retry-based strategies (*retry_modified_params*, *backtrack*) suggest a reactive error-handling pattern – responding to failures after they occur rather than anticipating and preventing them.

2. **Outlier masks structure:** Llama's extreme behavior obscured two meaningful differences (*change_strategy*, *skip_and_continue*) that distinguish top from bottom performers among the remaining eight models.

3. **Core differentiators are robust:** The three most predictive strategies of high performance – *verify_prerequisites*, *break_into_steps*, and *lookup_correct_value* – are unaffected by Llama's inclusion.

4. **Retry without modification is universally poor:** The *retry_same* strategy remains the only robustly underrepresented behavior among top performers, suggesting that the ability to recognize when repetition is futile is a consistent marker of effective agents.

A.8 ReAct Prompts

A.8.1 Exploration ReAct prompt

ReAct system prompt (with API docs).

```
REACT_SYSTEM_PROMPT_WITH_API_DOCS = """You are
↪  an AI assistant that completes tasks by
↪  interacting with APIs via bash commands.

## Current Session
- **Service**: {service_name}
- **Base URL**: {base_url}
- **Description**: {service_description}

## Environment
```

```
- You are authenticated as a user in the
↪  {service_name} workspace/account.
- Authentication is handled automatically via
↪  proxy. Use placeholder tokens like
↪  `<TOKEN>` where credentials would go.
- You execute bash commands (primarily curl) to
↪  interact with the {service_name} API.
- The environment is stateless between commands
↪  - you cannot install packages or persist
↪  files.

## Response Format
You must respond using XML tags. Think
↪  step-by-step, then execute a command OR
↪  declare completion.

**To execute a bash command:**
<thinking>
Your reasoning about what needs to be done and
↪  why this command will help.
</thinking>

<action>
Your bash command here (e.g., curl request)
</action>

**When the task is complete:**
<thinking>
Your reasoning confirming the task is done
↪  based on API responses.
</thinking>

<done>
Brief summary of what was accomplished.
</done>

## Rules
1. Execute ONE command at a time, then wait for
↪  the result.
2. Parse API responses carefully - extract IDs
↪  and data needed for subsequent calls.
3. If a command fails, analyze the error and
↪  try a different approach.
4. Only use <done> when the task is fully
↪  completed (not just when you've gathered
↪  information).

## API Documentation
{api_docs}
"""
```

A.8.2 Execution ReAct prompt

ReAct system prompt (with API docs).

```
REACT_SYSTEM_PROMPT = """You are an AI
↪  assistant that completes tasks by
↪  interacting with APIs via bash commands.

## Current Session
- **Service**: {service_name}
- **Base URL**: {base_url}
- **Description**: {service_description}

## Environment
- You are authenticated as a user in the
↪  {service_name} workspace/account.
- Authentication is handled automatically via
↪  proxy. Use placeholder tokens like
↪  `<TOKEN>` where credentials would go.
- You execute bash commands (primarily curl) to
↪  interact with the {service_name} API.
- If you are not sure how to use {service_name}
↪  API, explore the endpoint, parameters, and
↪  learn how it works.
- The environment is stateless between commands
↪  - you cannot install packages or persist
↪  files.

## Response Format
You must respond using XML tags. Think
↪  step-by-step, then execute a command OR
↪  declare completion.

**To execute a bash command:**
<thinking>
Your reasoning about what needs to be done and
↪  why this command will help.
</thinking>

<action>
Your bash command here (e.g., curl request)
</action>

**When the task is complete:**
<thinking>
Your reasoning confirming the task is done
↪  based on API responses.
</thinking>

<done>
Brief summary of what was accomplished.
</done>

## Rules
1. Execute ONE command at a time, then wait for
↪  the result.
2. Parse API responses carefully - extract IDs
↪  and data needed for subsequent calls.
3. If a command fails, analyze the error and
↪  try a different approach.
4. Only use <done> when the task is fully
↪  completed (not just when you've gathered
↪  information).

"""
```

Table 9: System prompt length (approximate tokens) by documentation condition and service.

Service	no_docs	relevant	all_docs
Box	380	3,230	22,320
Calendar	450	9,980	22,390
Linear	390	6,340	22,330
Slack	380	3,890	22,330

A.9 Model Details

Model (OpenRouter ID)	Temp.	Context	In $/M	Out $/M
google/gemini-3-flash-preview	Default	1,048,576	$0.50	$3.00
deepseek/deepseek-v3.2	Default	163,840	$0.25	$0.38
x-ai/grok-4.1-fast	Default	2,000,000	$0.20	$0.50
openai/gpt-oss-120b	Default	131,072	$0.039	$0.19
moonshotai/kimi-k2-0905	Default	262,144	$0.39	$1.90
meta-llama/llama-4-scout	Default	327,680	$0.08	$0.30
qwen/qwen3-235b-a22b-2507	Default	262,144	$0.071	$0.10
mistralai/devstral-2512	Default	262,144	$0.05	$0.22
anthropic/claude-haiku-4.5	Default	200,000	$1.00	$5.00

Table 10: Models and inference settings used in our experiments. All models were accessed via the OpenRouter API using each provider's default temperature.

A.10 Sandbox Architecture

A.10.1 Request Interception and Routing

Agent code executes within a Code Executor that intercepts outbound HTTP requests and rewrites production URLs to sandbox endpoints. For example, a request to `https://slack.com/api/chat.postMessage` is transparently rewritten to:

```
{BASE_URL}/api/env/{env_id}/services/slack/chat.postMessage
```

This interception occurs at the HTTP client level, requiring no modification to agent code – agents believe they are communicating with production APIs.

An Isolation Middleware layer extracts the environment identifier from the request path and establishes a database session scoped to the corresponding schema. All downstream handlers operate on this isolated session, ensuring that concurrent environments cannot observe or mutate each other's state.

A.10.2 Database-Backed State

We represent service state as a PostgreSQL database, with one schema per environment instance. Entity types map directly to tables; for example, Slack's state includes `users`, `channels`, `messages`, `reactions`, and `files` tables. Relationships (e.g., message → channel) are enforced via foreign keys.

Schemas are derived from service specifications:

- **REST APIs** (Slack, Box, Calendar): We parse OpenAPI schemas and SDK type definitions to generate SQLAlchemy ORM models. Nullable fields, enumerations, and nested objects are preserved.

- **GraphQL APIs** (Linear): We traverse the GraphQL schema, inferring relational structure from type definitions. Nested types are denormalized into separate tables with foreign key relationships.

A.10.3 Environment Lifecycle

Each benchmark task executes in a fresh environment:

1. **Schema Creation**: A new PostgreSQL schema `state_{env_id}` is created by cloning the structure from a service-specific template schema.

2. **Data Seeding**: Initial state is populated via `INSERT ... SELECT` from the template, ensuring atomic, deterministic initialization.

3. **Execution**: The agent interacts with replica APIs; all mutations modify tables within the isolated schema.

4. **Snapshotting**: Before and after execution, we create snapshot tables (`{table}_snapshot_before`, `{table}_snapshot_after`) to preserve state for evaluation.

5. **Cleanup**: Schemas are dropped after evaluation, reclaiming resources.

A.10.4 Replica API Implementations

We implement API handlers that read from and write to the isolated database session:

REST Services (Slack, Box, Google Calendar). Handlers are organized as a dispatch table mapping endpoints to functions. Each handler validates the request against the API specification, performs the corresponding database operations, and returns a response matching the documented schema. For example, `conversations.create` inserts a row into the `channels` table, adds the creator to `channel_members`, and returns the channel object.

GraphQL Services (Linear). We use a schema-first approach with resolver functions bound to each field. The GraphQL schema is loaded from Linear's public schema definition. Queries traverse the database via SQLAlchemy relationships; mutations perform validated writes and return updated objects.

Both implementations handle authentication tokens (injected via environment variables), pagination, rate limiting responses, and error conditions documented in the respective API specifications.

A.10.5 State Diffing

Evaluation relies on computing the precise difference between pre- and post-execution states. Given snapshot tables T_{before} and T_{after} for each entity type (where T denotes a table snapshot of that entity's state), we compute INSERTS, DELETES, and UPDATES as defined in Section 3.1. Set differences are computed via SQL `EXCEPT` queries; updates are detected by joining on primary keys and comparing column values. The resulting `DiffResult` structure contains all inserted, deleted, and modified entities, which is then compared against the task's expected state changes.

Table 11: API schema sources, licenses, and access dates.

Service	Type	Schema source	License	Date
Slack	REST	https://api.slack.com/methods	MIT	2026-02-02
Box	REST	https://developer.box.com/reference/	Apache 2.0	2026-01-25
Calendar	REST	https://developers.google.com/calendar/api/v3/reference	Apache 2.0	2026-02-02
Linear	GraphQL	https://studio.apollographql.com/public/Linear-API/	MIT	2026-02-02

For each service, we validated replica responses against production by issuing identical API calls to both systems and comparing response structure (JSON keys, types, pagination format, HTTP status codes).

Licensing. All API specifications used in Agent-Diff are released under permissive open-source licenses: Slack and Linear under MIT; Box and Google Calendar under Apache 2.0. These licenses permit reproduction, modification, and redistribution, including the creation of derivative works such as our replica implementations. Agent-Diff benchmark code and data are released under MIT at https://github.com/agent-diff/agent-diff.

A.11 Task Generation Prompt

The generation pipeline proceeds in two stages. First, an algorithmic sampler draws the task horizon $n \sim \text{TruncExp}(\lambda{=}0.3,\ 1..40)$ and samples n endpoints i.i.d. uniform from the target service's endpoint set $\mathcal{E}_s$. These, together with the full API schema and an optional human-authored seed template, are injected into the generation prompt. The generation model (Claude Opus 4.5 or Gemini 3) then produces the user-facing task prompt, declarative assertions, and – when no human seed is provided – the initial database state. The example below shows the Google Calendar variant; Slack, Box, and Linear variants substitute the corresponding schema and endpoints.

Task generation prompt example (Google Calendar).

```
You are a benchmark task designer for an API
↪  agent evaluation
suite. Your goal is to create a realistic,
↪  multi-step task that
requires an agent to interact with the Google
↪  Calendar API.

== INPUTS (provided by the system) ==

TARGET SERVICE: Google Calendar
API SCHEMA:
{api_schema}

SAMPLED TASK HORIZON: n = {n}

SAMPLED ENDPOINT SEQUENCE (with replacement):
{endpoint_sequence}

HUMAN-AUTHORED SEED TEMPLATE (may be empty):
{seed_template_or_empty}

== YOUR TASK ==

Using the inputs above, follow these steps:

STEP 1: Generate character names.
Pick a number m in [1, 6]. Generate m unique
↪  human names drawn
from diverse cultural traditions (e.g.,
↪  Japanese, Iranian,
American, Indian, Ukrainian, Polish, Nigerian,
↪  Korean,
Brazilian). These are the names you may use in
↪  the
scenario.

STEP 2: Design the action sequence.
Create a coherent sequence of steps that
↪  requires ALL {n}
sampled endpoints in the order given
↪  (reordering is allowed
where dependencies require it). For each step:
- Specify which sampled endpoint it maps to
- Justify why this step is needed in the
↪  scenario
- Explain how it connects to adjacent steps
```

```
IMPORTANT creative guidelines:
- Do NOT default to corporate/meeting scenarios
- Choose a vivid, original theme (e.g.,
↪  astronomy club,
  community garden, film festival, cooking
  ↪  competition,
  volunteer coordination, music rehearsals, pet
  ↪  adoption)
- Calendar and event names should be specific
↪  and colorful
  (e.g., "Perseid Meteor Shower Watch Party"
  ↪  not "Team
  Meeting")
- Include realistic complications: scheduling
↪  conflicts,
  permission sharing, event updates,
  ↪  cancellations

STEP 3: Determine information availability.
For each step, decide whether required
↪  identifiers (calendar
IDs, event IDs, user emails) should be:
(a) Provided explicitly in the prompt, OR
(b) Discoverable only via API queries (e.g.,
↪  listing
    calendars to find the right one by name)

STEP 4: Generate the user-facing prompt.
Write a natural-language task prompt that:
- Does NOT reveal the API call sequence directly
- Requires the agent to INFER the necessary
↪  steps
- Uses conversational language (as if a real
↪  user is asking
  their AI assistant)
- Embeds implicit requirements mapping to your
↪  action sequence

STEP 5: Generate or validate the seed state.
If a HUMAN-AUTHORED SEED TEMPLATE was provided
↪  above, validate
that it supports the task and note any required
↪  additions.
If no seed was provided, generate the initial
↪  database state:
- Calendars and their properties
- Events, attendees, and scheduling details
- Users and email addresses
- ACL (access control) entries
Use the character names from Step 1. Include
↪  distractor
entities that the agent must navigate around.

STEP 6: Generate assertions.
For each expected state change, write a
↪  declarative assertion
in the following format:
- diff_type: inserted / updated / deleted
- entity: table name
- where: field-level predicates identifying the
↪  record
- expected_count: number of matching rows

Also list fields to IGNORE (e.g., updated_at
↪  timestamps,
auto-generated IDs) so that non-deterministic
↪  changes do not
cause false negatives.
```

A.12 Schema Derivation and Replica Validation

Why replication is feasible. All information required to construct behaviorally equivalent replicas $\hat{f}$ is publicly available:

1. **Entity schemas**: Services publish complete type definitions via OpenAPI specifications (Slack (Slack Technologies, 2026b), Box (Box, Inc., 2026a), Google Calendar (Google LLC, 2026)) or GraphQL introspection (Linear (Linear Inc., 2026a)). These define the structure of S.

2. **Operation semantics**: API documentation specifies the mapping from requests $\mathcal{R}$ to state changes ΔS. For example, Slack's `chat.postMessage` documentation fully specifies required parameters, validation rules, and the resulting `Message` entity.

3. **Response formats**: SDK type definitions (e.g., `python-slack-sdk` (Slack Technologies, 2026a), `box-python-sdk` (Box, Inc., 2026b), `@linear/sdk` (Linear Inc., 2026b)) provide exact response schemas for $\mathcal{O}$.

4. **Error conditions**: HTTP status codes and error payloads are documented; undocumented edge cases can be systematically discovered via controlled probing of production APIs.

Schema sources. For REST APIs (Slack, Box, Google Calendar), we derive PostgreSQL schemas from OpenAPI specifications and official SDK type definitions. For GraphQL APIs (Linear), we parse the introspection schema and infer relational structure, denormalizing nested entities into separate tables with foreign key relationships.

Entity–table mapping. Formally, let $\mathcal{E} = \{E_1, \ldots, E_n\}$ denote entity types (e.g., `User`, `Channel`, `Message` for Slack). The complete state is $S = \bigcup_{E_i \in \mathcal{E}} \text{Table}(E_i)$, with each table's schema derived from the service's public type definitions.

Replica fidelity validation. We employ two complementary strategies: (1) *automated conformance testing* – for each implemented endpoint, we issue identical requests to both the production API and our replica, then assert structural equivalence of JSON responses (matching keys, types, pagination format, and error codes); and (2) *adversarial probing* – an auxiliary LLM agent systematically

exercises edge cases (invalid parameters, missing fields, boundary values) against production APIs to surface undocumented error conditions, which we then replicate. Schema sources, access dates, and endpoint coverage are listed in Table 11.

A.13 Error and Recovery Strategy Methodology

This appendix documents the complete taxonomy used for qualitative coding of agent execution traces, the annotation procedure, and how fine-grained categories were merged for analysis.

A.13.1 Error Annotation

Fine-Grained Error Taxonomy Each agent execution trace was evaluated for 32 fine-grained error types organized into four domains.

Tool Use Errors. Errors related to how the agent interacts with tools and APIs.

endpoint_selection
Incorrect or irrelevant API endpoint choices for the intended operation.

parameter_misuse
Wrong parameter names, wrong data types, wrong structure (missing required keys, extra nesting), using a field not accepted by the tool, or mapping data to the wrong field when a more appropriate field exists.

format_errors
Unparseable or malformed tool output: invalid JSON, truncation, or mixing natural language into machine-readable payloads.

code_errors
Valid tool calls that fail during execution: Bash syntax errors, runtime exceptions (NameError, ImportError), logic bugs, or environment misconceptions.

Hallucination Errors. Errors where the agent fabricates or asserts invented information as truth, distinct from reasoning errors (logic failures) and assumption errors (guessing without checking).

parameter_hallucination
Agent asserts invented parameter values (IDs, names, timestamps, URLs) as truth, when not grounded in the trace or user input.

outcome_hallucination
Agent falsely claims task completion or success despite evidence showing the task was not completed.

state_hallucination
Agent fabricates state that doesn't exist (e.g., "the file was created" when it wasn't). Distinct from *state_tracking_error* which involves forgetting existing state.

action_hallucination
Agent claims to have performed an action that doesn't appear in the trace – fabricating that execution occurred.

capability_hallucination
Agent believes a tool/API can do something it cannot, or invents non-existent endpoints.

context_hallucination
Agent references information not present in trace, prompt, or API responses.

other_hallucination
A hallucination not covered by the categories above.

Reasoning Errors. Errors involving logic failures, memory issues, or flawed inference – concerning how the agent thinks rather than fabricating information.

time_orientation_error
Confusing past vs. future events, incorrect date/time calculations, or timezone confusion.

state_tracking_error
Agent forgot previous actions, failed to update understanding after new information, or repeated already-completed actions (memory failure).

goal_misalignment_error
Solving a different problem than asked, missing implicit requirements, or understanding a requirement but deprioritizing it.

causal_reasoning_error
Misattributing why something failed, reversing cause and effect, or missing causal steps.

confirmation_bias
Ignoring contradictory error messages, persisting with failing approach despite clear feedback.

logical_fallacy
False dichotomy, circular reasoning, or non sequitur conclusions.

assumption_error
Agent guessed defaults or API behavior without checking first.

negation_error
Inverting boolean conditions, misunderstanding "not"/"except"/"exclude".

scope_generalization_error
Over-generalizing from specific instructions or being too literal.

dependency_ordering_error
Performing actions in wrong sequence (about ordering, not missing steps).

incomplete_execution_error
Agent planned required subtasks but failed to attempt some entirely (about omission).

premature_termination_error
Agent stops and concludes task is complete before finishing, without recognizing remaining work.

quantitative_reasoning_error
Off-by-one errors, unit confusion, incorrect aggregation or sorting.

reference_resolution_error
Misunderstanding what "it"/"this" refers to, confusing similar entities.

instruction_fidelity_error
Modifying content that should be preserved verbatim or adding unrequested embellishments.

reasoning_action_mismatch
Agent's explicit reasoning contradicts the action it executes.

infinite_loop_error
Agent stuck in pathological loop, repeating identical actions without progress.

other_reasoning_error
A reasoning error not covered above.

Other Categories.

model_refusal
Agent refuses to act, asks user for information it could retrieve, or delegates execution back to user.

other_error
An error that doesn't fit any category above.

Merged Error Categories For interpretability, the 32 fine-grained error annotations were consolidated into 6 merged categories:

1. **Endpoint Selection** – calling a non-existent or semantically wrong endpoint (direct mapping from *endpoint_selection*).
2. **Parameter Errors** – incorrect values, missing required fields, type mismatches, or hallucinated parameters. Merged from: *parameter_misuse* ∨ *parameter_hallucination*.
3. **Execution Errors** – malformed JSON, syntax errors, or runtime exceptions in generated code. Merged from: *format_errors* ∨ *code_errors*.
4. **Hallucination** – fabricating API responses, system state, actions, or task outcomes (excluding parameter hallucination, captured above). Merged from: *outcome_hallucination* ∨ *state_hallucination* ∨ *action_hallucination* ∨ *capability_hallucination* ∨ *context_hallucination* ∨ *other_hallucination*.
5. **Reasoning Failures** – flawed planning, goal misalignment, confirmation bias, or incorrect state tracking. Merged from 16 reasoning error subtypes (excluding *incomplete_execution_error* and *premature_termination_error*).
6. **Incomplete Execution** – runs where the agent stopped before completing all required subtasks or declared success prematurely. Merged from: *incomplete_execution_error* ∨ *premature_termination_error*.

Categories are not mutually exclusive; a single run may exhibit multiple error types.

A.13.2 Recovery Strategy Annotation

Recovery strategies are behavioral patterns agents exhibit when encountering errors or obstacles during task execution. We identified 14 distinct strategies plus a "no recovery" indicator:

retry_same
Retried the exact same action unchanged, hoping for a different result.

retry_modified_params
Retried with adjusted parameters (different ID, format, value).

switch_tool
Switched to a different tool/endpoint to achieve the same goal.

lookup_correct_value
Searched or queried to find the correct ID/name/value.

backtrack
Returned to an earlier step to gather missing information.

parse_error_message
Extracted useful information from error output to inform next action.

handle_ui_obstacle
Handled popup, dialog, login wall, or similar UI blocker.

change_strategy
Abandoned current approach entirely and tried a different method.

break_into_steps
Decomposed a complex action into smaller sequential steps.

verify_prerequisites
Checked if required conditions were met before retrying.

skip_and_continue
Moved past a blocking item to complete other parts of the task.

wait_and_retry
Added delay for rate limits or async operations.

use_fallback
Used a secondary/backup method when primary failed.

other_recovery_strategy
A recovery strategy not covered by the categories above.

no_recovery_attempted
Agent gave up immediately or got stuck in a loop without any recovery attempt.

Each strategy is coded as binary (present/absent) per run, allowing a run to exhibit multiple strategies.

A.13.3 Annotation Procedure

Each of the 4,032 traces was annotated by an LLM judge (Gemini-3-Flash) using a structured JSON schema. For each error and recovery category, the judge provides an explicit `present/absent` judgment with a concrete trace excerpt as evidence. The judge receives each trace alongside a reference trace – the highest-scoring run on the same task – to ground its assessments.

Normalization. Prevalence aggregates are computed as the percentage of runs exhibiting each error or strategy, stratified by model. For comparative analyses (top vs. bottom performers), we apply paired clustered Bayesian bootstrap to control for task difficulty (see Appendix A.7.4).

A.13.4 Judge Prompt

The trace annotation judge receives three inputs: (1) the trace under analysis (`RUN_TO_ANALYZE`), (2) a reference trace for comparison (`BEST_RUN` or `ONE_OF_THE_BEST_RUNS`), and (3) metadata including the scores of both runs. The judge is instructed to evaluate seven categories in order, returning structured JSON with explicit `present/absent` judgments and concrete trace excerpts for every subcategory.

System message.

```
You are a helpful assistant. You are required
↪  to analyze
RUN_TO_ANALYZE and compare it to
↪  {other_run_type} as a reference.
```

Context preamble. The human message includes the run under analysis, a reference run with score context, and the following evaluation instructions:

```
IMPORTANT EVALUATION CONTEXT:
The agents being evaluated were given these
↪  instructions in their
system prompt:
- "Use execute_bash to interact with [Service]
↪  API at [endpoint].
   Complete the task using the tools provided."
- "Authentication is handled automatically via
↪  proxy. Leave a
   placeholder credential where you would add a
   ↪  real token."

Therefore, do NOT flag as errors:
- Agent not explicitly handling authentication
↪  (it's automatic
  via proxy)
- Agent using placeholder credentials or
↪  assuming auth works
- Agent proceeding directly to API calls
↪  without auth setup

Now analyze RUN_TO_ANALYZE (and use
↪  {other_run_type} only as a
reference for what a correct approach looks
↪  like). Evaluate the
following categories in order. For each
↪  category, provide the
required fields as specified.
```

Category 1: Tool Use Errors.

```
1) Tool Use Errors
Errors related to how the agent interacts with
↪  tools and APIs.
Evaluate each subtype explicitly:

endpoint_selection:
Determine whether the agent consistently
↪  selects correct endpoints.
- present: True if there are any incorrect or
↪  irrelevant endpoint
  choices
- explanation: Brief summary of the issue (or
↪  why none were found)
- example: One concrete example from the trace
↪  (or 'N/A')

parameter_misuse:
Determine whether the agent ever calls an API
↪  endpoint with
incorrectly formatted, incorrectly typed, or
↪  semantically
misplaced parameters. This includes:
- Wrong parameter names
- Wrong data types (string vs int vs list vs
↪  object)
- Wrong structure (scalar where object required,
↪  missing keys)
- Using a field not accepted by the tool
- Semantic field mapping errors: putting data
↪  in the wrong field
  (e.g., location info in summary instead of
  ↪  location field)
Returns: present, explanation, example.
```

```
format_errors:
Determine whether the agent ever produces
↪  tool-related output
that is unparseable or malformed. Includes
↪  invalid JSON, incorrect
nesting, mixing natural language into
↪  machine-readable payloads,
truncation, or any syntax/format issue causing
↪  parse failure.
Returns: present, explanation, example.

code_errors:
Determine whether the agent produces valid tool
↪  calls but the
content fails during execution. Includes Bash
↪  syntax errors,
runtime exceptions (NameError, ImportError),
↪  logic bugs, or
environment misconceptions.
Returns: present, explanation, example.
```

Category 2: Model Refusal.

```
2) Model Refusal
Determine whether the agent refuses to perform
↪  the task, asks the
user for information it could retrieve itself,
↪  OR delegates
execution back to user. This includes:
- Explicitly refusing to perform the task
- Asking user for IDs, tokens, or file contents
↪  the agent could
  find itself
- Passive delegation: gathering information but
↪  providing
  recommendations instead of performing actions
Returns: present, explanation, example.
```

Category 3: Hallucination Errors.

```
3) Hallucination Errors
Hallucinations are when the agent FABRICATES or
↪  ASSERTS invented
information as truth. Distinct from reasoning
↪  errors (logic
failures) and assumption errors (guessing
↪  without checking).

For EACH type, explicitly evaluate whether it
↪  occurred. You MUST
provide a judgment (present: true/false) and
↪  example for EVERY
category:

- parameter_hallucination: Agent ASSERTS
↪  invented parameter values
  (IDs, names, timestamps, URLs) as truth, not
  ↪  grounded in trace
  or user input.
- outcome_hallucination: Agent falsely CLAIMS
↪  task completion
  despite evidence showing the task was not
  ↪  completed.
- state_hallucination: Agent FABRICATES state
↪  that doesn't exist.
  Distinct from state_tracking_error
  ↪  (FORGETTING existing state).
- action_hallucination: Agent CLAIMS to have
↪  performed an action
  that doesn't appear in the trace.
- capability_hallucination: Agent believes a
↪  tool/API can do
  something it cannot, or invents non-existent
  ↪  endpoints.
- context_hallucination: Agent references
↪  information not present
  in trace, prompt, or API responses.
- other_hallucination: A hallucination not
↪  covered above.

Returns per category: present, example,
↪  explanation (only for
other_hallucination).
```

Category 4: Reasoning Errors.

```
4) Reasoning Errors
Reasoning errors involve logic failures, memory
↪  issues, or flawed
inference. About HOW the agent thinks, not
↪  fabricating information.

IMPORTANT DISTINCTIONS:
- state_tracking_error = agent FORGETS (memory
↪  failure)
- state_hallucination = agent INVENTS
↪  (fabrication)
- assumption_error = agent GUESSES without
↪  checking
- hallucination = agent ASSERTS invented facts
↪  as known truth
- dependency_ordering_error = wrong SEQUENCE
- incomplete_execution_error = OMITS steps
↪  entirely
- premature_termination_error = STOPS early
↪  thinking task is done

For EACH type, provide present: true/false and
↪  example:

- time_orientation_error: Confusing past vs
↪  future, incorrect
  date/time calculations, timezone confusion.
- state_tracking_error: Agent FORGOT previous
↪  actions, failed to
  update understanding, or repeated completed
  ↪  actions.
- goal_misalignment_error: Solving different
↪  problem than asked,
  missing implicit requirements, deprioritizing
  ↪  explicit ones.
- causal_reasoning_error: Misattributing why
↪  something failed,
  reversing cause and effect.
- confirmation_bias: Ignoring contradictory
↪  error messages,
  persisting with failing approach despite
  ↪  clear feedback.
- logical_fallacy: False dichotomy, circular
↪  reasoning, non
  sequitur conclusions.
- assumption_error: Agent GUESSED defaults or
↪  API behavior
  WITHOUT CHECKING first.
- negation_error: Inverting boolean conditions,
↪  misunderstanding
  "not"/"except"/"exclude".
- scope_generalization_error: Over-generalizing
↪  from specific
```

```
  instructions, being too literal.
- dependency_ordering_error: Performing actions
↪  in wrong SEQUENCE.
- incomplete_execution_error: Planned subtasks
↪  but failed to
  attempt some entirely (OMISSION).
- premature_termination_error: Stops and
↪  concludes task is
  complete before finishing all required steps.
- quantitative_reasoning_error: Off-by-one
↪  errors, unit
  confusion, incorrect aggregation or sorting.
- reference_resolution_error: Misunderstanding
↪  what "it"/"this"
  refers to, confusing similar entities.
- instruction_fidelity_error: Modifying content
↪  that should be
  preserved verbatim or adding unrequested
  ↪  embellishments.
- reasoning_action_mismatch: Agent's reasoning
↪  contradicts the
  action it actually executes.
- infinite_loop_error: Stuck in pathological
↪  loop, repeating
  identical actions without progress.
- other_reasoning_error: Not covered above.
```

Category 5: Recovery Strategies.

```
5) Recovery Strategies
For EACH type, evaluate whether the agent
↪  attempted it. Provide
present: true/false and example for EVERY
↪  category:

- retry_same: Retried exact same action
↪  unchanged.
- retry_modified_params: Retried with adjusted
↪  parameters.
- switch_tool: Switched to different
↪  tool/endpoint for same goal.
- lookup_correct_value: Searched/queried to
↪  find correct value.
- backtrack: Returned to earlier step for
↪  missing information.
- parse_error_message: Extracted useful info
↪  from error output.
- handle_ui_obstacle: Handled popup, dialog, or
↪  login wall.
- change_strategy: Abandoned current approach
↪  entirely.
- break_into_steps: Decomposed complex action
↪  into smaller steps.
- verify_prerequisites: Checked conditions
↪  before retrying.
- skip_and_continue: Moved past blocking item
↪  to complete others.
- wait_and_retry: Added delay for rate limits
↪  or async ops.
- use_fallback: Used secondary/backup method.
- other_recovery_strategy: Not covered above.
- no_recovery_attempted: Gave up or stuck in
↪  loop without
  recovery.
```

Category 6: Other Errors.

```
6) Other Errors
Determine if there are errors not covered by
↪  categories 1-4.
Returns: present, explanation (including
↪  proposed subcategory
name), example.
```

Category 7: Qualitative Summary.

```
7) Qualitative Summary
Provide a high-level narrative analysis of this
↪  run.

Scoring dimensions (each 0--5):
- planning_score: Action sequencing,
↪  adaptation, efficiency.
  5=Excellent (clear, efficient, proactive)
  3=Mixed (progress with avoidable detours)
  0=Non-functional (no meaningful plan)
- reasoning_score: Correctness of inferences,
↪  use of context.
  5=Excellent (all inferences correct,
  ↪  self-corrects)
  3=Mixed (notable logic errors)
  0=Non-functional (incoherent reasoning)
- tool_use_score: Endpoint selection, parameter
↪  formatting.
  5=Excellent (correct endpoints, proper
  ↪  formatting)
  3=Mixed (some incorrect endpoints or
  ↪  malformed params)
  0=Non-functional (cannot interact with APIs)
- recovery_score: Failure detection, root cause
↪  diagnosis.
  5=Excellent (quick detection, effective
  ↪  correction)
  3=Mixed (slow diagnosis, wastes iterations)
  0=Non-functional (ignores all error signals)
- hallucination_score: Resistance to
↪  fabrication (5=none, 0=extreme).
  5=None (all claims grounded)
  3=Moderate (some fabricated details, core
  ↪  reasoning sound)
  0=Extreme (nearly all assertions fabricated)

Each score includes a brief justification
↪  (*_explanation).

Narrative fields:
- overall_description: 2-3 sentence summary of
↪  what went wrong.
- key_insight: Single most important takeaway.
- model_behavior_pattern: How the model
↪  approaches this task type.
- implications_for_reliability: What this
↪  reveals about
  robustness.
- worthy_example: true/false (interesting for
↪  qualitative
  analysis).
- why_worthy_example: Explanation if worthy,
↪  else 'N/A'.

Return results in the required structured JSON
↪  format.
```

Category	Model / Service	Score	Root cause	Trace summary
Endpoint selection	`gpt-oss-120b` *Linear*	50%	Queries `issueStates`, which does not exist. The API replies "Did you mean workflowStates?" This is a clean endpoint-selection error.	The agent queries `issueStates` (step 6), receives "Did you mean workflowStates?", self-corrects to `workflowStates` (step 7), but then fails repeatedly on shell escaping for the mutation (steps 8–10).
Parameter errors	`gpt-oss-120b` *Linear*	0%	Uses `priority: 3` (Medium) when the task requires High priority, which should be 2. This is a simple concrete value error.	After retrieving the team ID, the agent attempts `issueCreate` with `priority: HIGH` (step 5) and receives "Int cannot represent non-integer value." It introspects the schema but gets null (step 7), then guesses `priority: 3` instead of the correct value 2.
Execution errors	`kimi-k2-0905` *Box*	0%	Missing `-L` on `curl` for Box file-content download. Box returns a 302 redirect, so `curl` gets empty content. One missing flag causes total failure.	The agent correctly navigates the folder hierarchy and locates the target CSV file (steps 1–3). However, every `/content` download returns empty because `curl` lacks `-L` to follow Box's 302 redirect (steps 4, 6, 7, 13). After 15 steps of failed workarounds, it renames the folder with placeholder data.
Hallucination	`gemini-3-flash-preview` *Slack*	20%	The API returns the real DM channel ID `DUPUX56JRI2`, but the model fabricates `D01L06A7W3N` in its reasoning and uses it, producing `channel_not_found`.	The agent successfully lists users and channels (steps 1–3), then opens a DM and receives channel ID `DUPUX56JRI2` (step 5). In step 7, it fabricates a different ID, `D01L06A7W3N`, in its reasoning and uses it, receiving `channel_not_found`.
Reasoning failures	`gpt-oss-120b` *Calendar*	87.5%	State-tracking error: the agent planned to delete the "Old Courier Shifts" calendar in step 1 but never executed the deletion. It proceeded as if the subtask were complete.	The agent outlines a multi-step plan including deleting a legacy calendar, creating new route calendars, and updating permissions. It executes the creation and permission steps correctly, but skips the deletion call entirely. In its final summary it reports all tasks complete and never notices the omission.
Incomplete execution	`devstral-2512` *Calendar*	40%	The agent spends 40 steps searching calendars one by one and runs out of steps before setting up any of the three required watches. All watch assertions fail.	Steps 1–3 correctly subscribe to a calendar, update its metadata, and patch an event. The agent then needs to find calendars with "coolcool-cool" events, but searches one by one across 60+ calendars (steps 5–40), exhausting its step budget before setting up any of the three required watches.

Table 12: Representative failure examples for each error category. Each entry is the single most illustrative run selected from 4,032 analyzed traces. Trace summaries are condensed from the full agent execution log.

Slack	Box	Calendar	Linear
`chat.delete`	`GET /collections`	`acl.delete`	`comments`
`chat.postMessage`	`POST /comments`	`acl.get`	`issueLabels`
`chat.update`	`DELETE /comments/{id}`	`acl.insert`	`issues`
`conversations.archive`	`PUT /comments/{id}`	`acl.list`	`teams`
`conversations.create`	`POST /files/content`	`acl.patch`	`users`
`conversations.history`	`DELETE /files/{id}`	`acl.update`	`workflowStates`
`conversations.info`	`GET /files/{id}`	`acl.watch`	`commentCreate`
`conversations.invite`	`PUT /files/{id}`	`calendarList.delete`	`commentDelete`
`conversations.join`	`GET /files/{id}/comments`	`calendarList.get`	`commentUpdate`
`conversations.kick`	`GET /files/{id}/content`	`calendarList.insert`	`issueCreate`
`conversations.leave`	`POST /files/{id}/content`	`calendarList.list`	`issueLabelCreate`
`conversations.list`	`GET /files/{id}/tasks`	`calendarList.patch`	`issueLabelUpdate`
`conversations.members`	`POST /folders`	`calendarList.update`	`issueRelationCreate`
`conversations.open`	`DELETE /folders/{id}`	`calendarList.watch`	`issueRelationDelete`
`conversations.rename`	`GET /folders/{id}`	`calendars.clear`	`issueUpdate`
`conversations.replies`	`PUT /folders/{id}`	`calendars.delete`	`teamCreate`
`conversations.setTopic`	`GET /folders/{id}/items`	`calendars.get`	`teamMembershipCreate`
`conversations.unarchive`	`GET /hubs`	`calendars.insert`	`workflowStateArchive`
`reactions.add`	`POST /hubs`	`calendars.patch`	`workflowStateCreate`
`reactions.remove`	`GET /hubs/{id}`	`calendars.update`	
`search.all`	`PUT /hubs/{id}`	`channels.stop`	
`search.messages`	`POST /hubs/{id}/manage_items`	`colors.get`	
`users.conversations`	`GET /search`	`events.delete`	
`users.info`	`POST /tasks`	`events.get`	
`users.list`	`DELETE /tasks/{id}`	`events.import`	
	`PUT /tasks/{id}`	`events.insert`	
	`GET /users/me`	`events.instances`	
		`events.list`	
		`events.move`	
		`events.patch`	
		`events.quickAdd`	
		`events.update`	
		`events.watch`	
		`freeBusy.query`	
		`settings.get`	
		`settings.list`	
		`settings.watch`	

Table 13: API endpoints in the benchmark. Total: 108 unique endpoints across 224 tasks.

A.14 Trace Comparison: Box Hub API

Model: `claude-haiku-4.5` **Service:** *Box* **Test:** Organize Research Hub

Prompt. *Create a new hub called* `Model Evaluations`. *Find all the JSON files in the* `agent-diff-research` *folder that contain model evaluation results and add them to this new hub.*

Without documentation (score = 0/8, 6 iterations).

Iteration 1 – Thinking.

I need to:

1. understand what a "hub" is in the Box API context,
2. find the `agent-diff-research` folder,
3. search for JSON files with model evaluation results,
4. create a new hub called `Model Evaluations`, and
5. add the JSON files to this hub.

Action.

```
curl -s https://api.box.com/2.0/folders/0 \
  -H "Authorization: Bearer <TOKEN>" | head -50
```

Observation.

```
{'stdout': '{"type": "folder", "id": "0",
↪  "sequence_id": null, "etag": null, "name":
↪  "All Files", ... }'}
```

Iteration 2 – Thinking.

The root folder contains `agent-diff-research` with ID `1173971943`. I next need to inspect that folder, find the JSON files, and create or verify the `Model Evaluations` hub.

Action.

```
curl -s "https://api.box.com/2.0/folders/117⌋
↪  3971943/items?limit=1000" \
  -H "Authorization: Bearer <TOKEN>" | jq '.'
```

Observation.

```
{'stdout': '{ "total_count": 8, "entries": [
  {"type": "file", "id": "8647156721",
   "name": "anthropic_claude-haiku-4.5_full_r⌋
   ↪  esults_20251208_081736.json"},
  ...
] }'}
```

Iteration 3 – Thinking.

The folder contains eight JSON files with model evaluation results. The agent now attempts to add them to a collection that it has not actually created or verified.

Action.

```
curl -s -X PUT "https://api.box.com/2.0/files⌋
↪  /8647156721" \
  -H "Authorization: Bearer <TOKEN>" \
  -H "Content-Type: application/json" \
  -d '{"collections": [{"id": "12345678"}]}' |
  ↪  jq '{id, name, collections}'
```

Observation.

```
{'stdout': '{ "id": null, "name": null,
↪  "collections": null }',
 'stderr': '', 'exit_code': 0}
```

Iteration 4 – Thinking.

The null response indicates that the earlier assumption about the collection was unsupported. The agent partially recognizes this inconsistency.

Action.

```
curl -s "https://api.box.com/2.0/collections"
↪  \
  -H "Authorization: Bearer <TOKEN>" | jq
  ↪  '.entries[] | {id, name}'
```

Observation.

```
{'stdout': '{ "id": "728834", "name":
↪  "Favorites" }',
 'stderr': '', 'exit_code': 0}
```

Iteration 5 – Thinking.

The agent notes that the observed collection is `Favorites`, not `Model Evaluations`, but does not recover correctly.

Action. No `<action>` or `<done>` tag produced.

Final summary.

The run incorrectly claims success: it states that a hub was created, eight files were added, and the result was verified, even though none of these claims is grounded in the observed API outputs.

With relevant documentation (score = 8/8, 7 iterations).

Iteration 1 – Thinking.

I should:

1. create a hub via `POST /hubs`,
2. find the `agent-diff-research` folder,

3. identify the JSON result files, and
4. add them using POST /hubs/{hub_id}/manage_items.

Action.

```
curl -X POST "https://api.box.com/2.0/hubs" \
  -H "Authorization: Bearer <TOKEN>" \
  -H "box-version: 2025.0" \
  -H "Content-Type: application/json" \
  -d '{
    "title": "Model Evaluations",
    "description": "Hub for model evaluation
    ↪  results"
  }'
```

Observation.

```
{'stdout':
↪  '{"type":"hubs","id":"614230230203",
"title":"Model Evaluations", ... }'}
```

Iteration 2 – Thinking.

The hub has been created successfully with ID 614230230203. The next step is to locate the target folder.

Action.

```
curl -X GET "https://api.box.com/2.0/search?q⌋
↪  uery=agent-diff-research&type=folder" \
  -H "Authorization: Bearer <TOKEN>"
```

Observation.

```
{'stdout': '{"total_count":1,"entries":[{"id⌋
↪  ":"1173971943",
"type":"folder","name":"agent-diff-research",
↪  ... }]}'}
```

Iteration 3 – Thinking.

The folder has ID 1173971943. The agent first tries a targeted search for evaluation files.

Action.

```
curl -X GET "https://api.box.com/2.0/search?q⌋
↪  uery=evaluation&type=file\
&file_extensions=json&ancestor_folder_ids=11⌋
↪  73971943&limit=200" \
  -H "Authorization: Bearer <TOKEN>"
```

Observation.

```
{'stdout': '{"total_count":0,"entries":[],"l⌋
↪  imit":200,"offset":0,
"type":"search_results_items"}', 'exit_code':
↪  0}
```

Iteration 4 – Thinking.

Because the targeted search fails, the agent falls back to listing the folder contents directly.

Action.

```
curl -X GET "https://api.box.com/2.0/folders/⌋
↪  1173971943/items?limit=1000" \
  -H "Authorization: Bearer <TOKEN>"
```

Observation.

```
{'stdout': '{"total_count":8,"entries":[
  {"type":"file","id":"8647156721",
   "name":"anthropic_claude-haiku-4.5_full_re⌋
   ↪  sults_20251208_081736.json"},
  ...
] }'}
```

Iteration 5 – Thinking.

The folder contains eight JSON files matching the expected result format, so the agent can add them to the hub.

Action.

```
curl -X POST "https://api.box.com/2.0/hubs/61⌋
↪  4230230203/manage_items" \
  -H "Authorization: Bearer <TOKEN>" \
  -H "box-version: 2025.0" \
  -H "Content-Type: application/json" \
  -d '{ "operations": [ ... ] }'
```

Observation.

```
{'stdout': '{"results":[
  {"status":"success","item":{"type":"file",⌋
  ↪  "id":"8647156721"}},
  {"status":"success","item":{"type":"file",⌋
  ↪  "id":"2466872085"}},
  ...
] }'}
```

Iteration 6 – Thinking.

All add operations succeeded. The final step is to verify the hub contents.

Action.

```
curl -X GET "https://api.box.com/2.0/hub_item⌋
↪  s?hub_id=614230230203&limit=200" \
  -H "Authorization: Bearer <TOKEN>" \
  -H "box-version: 2025.0"
```

Observation.

```
{'stdout': '{"entries":[
  {"type":"file","id":"8647156721",
   "name":"anthropic_claude-haiku-4.5_full_re⌋
   ↪  sults_20251208_081736.json"},
  {"type":"file","id":"2466872085",
   "name":"anthropic_claude-opus-4.5_full_res⌋
   ↪  ults_20251210_220420.json"},
  ...
] }'}
```

Final summary.

This run succeeds because the agent uses the documented hub endpoints directly, retrieves the correct folder, identifies the eight JSON result files, adds them to the hub, and verifies the final state through a follow-up query.

A.15 Benchmark Test Listing

A.15.1 Overall Benchmark Summary

Table 14: Benchmark task distribution by service and taxonomy dimension.

Metric	Box	Slack	Linear	Cal.	Total
Tasks	48	59	57	60	224
n^* (range)	1–13	1–14	1–13	1–24	1–24
n^* (mean)	4.6	5.6	5.2	5.9	5.3
Operation profile (non-exclusive, % of tasks)					
Search	92	64	89	77	80
Create	58	88	63	78	73
Read	54	68	14	82	55
Update	62	37	70	93	66
Delete	19	24	7	53	26
Entity scope					
Single-entity	28	33	33	11	105
Multi-entity	20	26	24	49	119
Information availability					
Explicit	6	36	25	10	77
Implicit	42	23	32	50	147
Prompt ambiguity					
Low	24	27	37	13	101
Medium	17	22	19	45	103
High	7	10	1	2	20

Benchmark summary. Table 14 shows that the benchmark contains 224 tasks distributed relatively evenly across the four services: 48 for Box, 59 for Slack, 57 for Linear, and 60 for Calendar. Task horizon is moderate overall ($n^* = 5.3$ on average), with Calendar containing the longest tasks (n^* range 1–24).

The benchmark also spans a broad range of operation types. Search is especially prominent in Box (92%) and Linear (89%), while create operations are most common in Slack (88%) and Calendar (78%). Update-heavy behavior is particularly characteristic of Calendar (93%) and Linear (70%), whereas delete operations are comparatively rare overall (26%) but substantially more common in Calendar (53%) than in the other services.

In terms of task structure, the benchmark is slightly skewed toward multi-entity scenarios (119 vs. 105 single-entity tasks), with Calendar contributing the largest share of multi-entity tasks. Most tasks also require some degree of information discovery: 147 tasks are implicit, meaning that key identifiers must be inferred or retrieved rather than read directly from the prompt. Finally, prompt ambiguity is concentrated in the low and medium categories (101 and 103 tasks, respectively), while only 20 tasks are labeled high ambiguity. Representative tasks for each service are shown in the tables below; the complete task inventory is provided in the supplementary material.

A.15.2 Representative Tasks

Table note. All appendix task tables (Tables 15, 16, 17, and 18) use the same columns: **#** is the task ID, **Svc** is the service, **Prompt** is the task instruction, d_{op} is the operation profile, d_{scope} is the entity scope, d_{info} is the information-availability category, d_{h} is the task horizon, and d_{amb} is the prompt-ambiguity category.

Full benchmark listing. The complete benchmark specification, including the full list of 224 tasks and their taxonomy annotations has been released with the benchmark repository.

Table 15: Representative tests for Slack.

#	Svc	Prompt	d_{op}	d_{scope}	d_{info}	d_{h}	d_{amb}
4	Slack	Create a new channel called 'rl-project'	C	s	exp	1	l
19	Slack	Search for all messages (4 messages) in the #engineering channel related to login issues and combine them into a single new message as DM to Hubert. Do not change the meaning of the original messages just combine them.	search+C+R	s	imp	4	m
31	Slack	Create a new channel called 'auth-force' and invite everyone who has posted about 'login' or 'password'.	search+C	m	imp	9	m
39	Slack	I need some help organizing our Diwali x Thanksgiving potluck celebration! We're doing a combined Indian and American traditions thing and I want to make sure we coordinate this properly across the team. First, can you check what channels we have that might be relevant for this event and see what's been discussed recently in #core-infra? I want to make sure I'm not stepping on any ongoing conversations. Also, I need to know who's on our team so I can figure out who to involve based on their backgrounds and expertise. Once you've got that context, please update the topics for #core-infra, #project-alpha, and #growth to reflect that we're planning this potluck celebration. Then post an announcement in #project-alpha about the event. I also need you to check who's currently in #growth to make sure the right people are included, and open a direct message with Kenji Sato since I need to coordinate with him separately about timing given APAC schedules. Oh, and there's an old message I posted earlier about the event that has wrong details - can you update it with the correct information? There's also an outdated announcement from last week that's no longer relevant, so please delete that. Finally, just react to Priya's message about bringing samosas to show I've seen it!	C+R+U+D	m	imp	13	h

Table 16: Representative Box tests.

#	Svc	Prompt	d_{op}	d_{scope}	d_{info}	d_{h}	d_{amb}
66	BOX	Create a new Box Hub titled 'Research Center'.	C	s	exp	1	l
76	BOX	Search for all plain-text files about Argentina's 2001 economic crisis. You should find two copies - one properly filed in the history folder and one misfiled in the root. Delete the misfiled copy, then read the correctly filed one. If it mentions 'Argentina', add the tag 'Latin_America' to it.	search+R+U+D	m	imp	4	m
85	BOX	Create a new hub called 'Model Evaluations'. Find all the JSON files in the agent-diff-research folder that contain model evaluation results and add them to this new hub.	search+C+R	m	imp	4	m
100	BOX	Your research institute's Box storage is disorganized. Somewhere in the archive, there are field research documents from cryptozoology expeditions — specifically sighting reports that may contain photographic evidence of unidentified creatures. Your task: Find a cryptozoology sighting report (search for relevant terms). Download and read its content. If the document mentions "photographic evidence" anywhere in the text, it should be tagged as verified; otherwise tag it unverified. Create a proper organizational structure: a main folder "Expeditions_2025" in the root, with a subfolder "Cryptid_Sightings" inside it. Move the sighting report into this subfolder with the appropriate tag. Add a comment to the file documenting your review: include today's date and the expedition name (which you'll find mentioned in the document's content). After moving the file, check its original location. If there are any obvious duplicate files (backup copies with similar names), delete them to clean up. Then rename the original source folder by appending "_archived" to its name. Finally, create a Hub called "2025 Field Research Index" and add the "Expeditions_2025" folder to it for easy access.	search+C+R+U+D	m	imp	11	h

Table 17: Representative tests for Linear.

#	Svc	Prompt	d_{op}	d_{scope}	d_{info}	d_{h}	d_{amb}
112	Linear	Add a comment to issue ENG-1 saying 'I am working on this now'	C	s	exp	1	l
123	Linear	Add the 'RL' label to the login issue that John Doe recently commented on.	search+R+U	s	imp	4	l
152	Linear	The IT Support team received a critical server outage report. Here's the workflow to execute: First, check if a label called "hardware-failure" exists. If it doesn't, create it. Create a new issue titled "Server rack B7 unresponsive - power supply failure" in the IT Support team. Apply the "hardware-failure" label to this ticket and assign it to Kofi for initial triage. Add a comment to the ticket with this diagnostic entry: "DIAG_LOG_001: Initial ping test failed. Checked physical connections. PSU indicator light is off. Replacement unit requested from inventory." Now update that same comment to append the following resolution note at the end: " ‖ UPDATE: PSU replaced at 14:32. Server responding. Monitoring for 24hrs." Finally, update the ticket to change the assignee from Kofi to Elena for post-incident verification, and move the ticket to "In Review" status.	search+C+R+U	m	imp	8	m
162	Linear	The PMO is conducting a Q1 resource allocation review. Here's what needs to happen: First, look at all teams and count how many members each team has. Find the team with the most members - this is our "fully staffed" benchmark. For every team that has FEWER members than the benchmark team, create a new issue in that team titled "Q1 Staffing Request - Need [X] additional team members" where [X] is the exact difference between that team's member count and the benchmark team's count. Set priority to High for these issues. Also, there's a misrouted issue: "API Documentation Update" was accidentally created in the Design team but belongs in Engineering. Move it to the Engineering team. Finally, add a comment to any issue in the Engineering team summarizing the analysis: "RESOURCE_AUDIT: Q1 staffing review complete. Engineering has [MAX] members (benchmark). Staffing gaps identified: Product needs [A], Design needs [B], QA needs [C]. Total headcount gap across org: [TOTAL]. Staffing request issues created in all understaffed teams." Replace the bracketed values with the actual numbers from your analysis. Note: [TOTAL] should be the sum of headcount gaps from ALL understaffed teams (not just Product, Design, and QA).	search+C+R+U	m	imp	12	h

Table 18: Representative tests for Calendar.

#	Svc	Prompt	d_{op}	d_{scope}	d_{info}	d_{h}	d_{amb}
167	Calendar	The guild needs organizing. First, remind me which calendars I have - I'm looking for our 'Dungeon Masters Guild' one. We're kicking off a new campaign called 'The Curse of the Crimson Dice' and I need to schedule Session Zero for Friday at 7pm on that calendar. The duration should be 3 hours. Amara (amara@test.com) offered to run a one-shot this weekend - find when she's free and schedule 'Amara's Epic One-Shot Adventure' for 4 hours at that time on the guild calendar (use Amara's timezone). Oh, and that Session Zero event needs more info - update the description to say 'Bring character concepts. Snacks provided. No phones at the table.' I want to see all the sessions we have planned this month on the guild calendar. Hiroshi (hiroshi@test.com) has been running great sessions and deserves to schedule his own games now - give him edit access to the Dungeon Masters Guild calendar. That old 'TPK Recovery Support Group (Postponed Indefinitely)' event is still sitting there as a bad joke from when we had that campaign wipe, and it should not be sitting there anymore. Finally, we've been mixing board game nights with RPG sessions and it's confusing people. Use the existing Board Game Bazaar calendar (ID cal_board_game_bazaar) for non-RPG gaming (create it if it doesn't exist). Then scan every game event on the Dungeon Masters Guild calendar: each event description is tagged 'Type: RPG' or 'Type: Non-RPG'. Copy every Non-RPG event to Board Game Bazaar; if it is recurring, copy it as a recurring event there too.	search+C+R+U+D	m	imp	15	m
172	Calendar	We're setting up the Time-Traveler's Convention and the timeline is fragile. First, check if 'Timeline Alpha' already exists in my calendars - it should be there. Create a new calendar called 'Timeline Beta' for our temporal experiments. Schedule the 'Paradox Prevention Seminar' for Tuesday June 19 at 10am, lasting 2 hours, on Timeline Beta. Sven (sven@test.com) is arriving from 2099 and needs to attend, but his arrival window is fluctuating - check his availability for Wednesday June 20 instead. Move the 'Paradox Prevention Seminar' to Sven's free slot on Wednesday. The Time Council has flagged the 'Grandfather Paradox Demonstration' as a Class 5 risk - find it and delete it immediately. Finally, grant Fatima (fatima@test.com) write access to Timeline Beta so she can document the changes to history.	search+C+R+U+D	m	imp	7	m
185	Calendar	Add a one-off event on my primary calendar called Emberglass Kiln Glow on June 25, 2018 from 7:00pm-8:30pm.	C	m	imp	1	l
186	Calendar	Please quick-add this to my primary calendar: 'Fogloom Archive Lantern Check on June 26, 2018 at 8:00pm for 45 minutes.' After it's created, fetch that event by ID so we can verify the parsed details. Also, set up a watch for changes to my calendar settings.	search+C+R+U	s	exp	4	m