# AGENTESCAPEBENCH: EVALUATING TOOL-GROUNDED DEPENDENCY REASONING IN LLM AGENTS

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

## ABSTRACT

As LLM-based agents increasingly rely on external tools, it is important to evaluate their ability to sustain tool-grounded reasoning beyond familiar workflows and short-range interactions. We introduce AGENTESCAPEBENCH, an escape-room-style benchmark that tests whether agents can infer, execute, and revise novel tool-use procedures under explicit long-range dependency constraints. Each task defines a directed acyclic dependency graph over tools and items, requiring agents to invoke real external functions, track hidden state revealed incrementally, propagate intermediate results, and submit a deterministically verifiable final answer. AgentEscapeBench includes 270 instances across five difficulty tiers and supports fully automated evaluation. Experiments with sixteen LLM agents and human participants show that performance drops sharply as the required interaction horizon and interleaved dependency complexity increase: humans decline from 98.3% success at difficulty-5 to 80.0% at difficulty-25, while the best model drops from 90.0% to 60.0%. Trajectory analysis attributes model failures mainly to breakdowns in long-range state tracking, clue adherence, and intermediate-result propagation. These findings suggest that current agents can often handle local tool use but still struggle with deep contextual dependencies. We hope AgentEscapeBench can serve as a diagnostic testbed for measuring tool-grounded dependency reasoning and informing future training efforts toward more robust state tracking, intermediate-result propagation, and interaction planning.

## 1 INTRODUCTION

Tool-calling has expanded the role of large language models (LLMs) beyond conversational question answering, enabling them to invoke APIs, execute code, manipulate files, and participate in real-world workflows. This shift has led to a new class of LLM-based agents with substantial practical potential (Anthropic, 2025). As such agents are increasingly deployed in production settings, rigorous and diagnostic evaluation of their capabilities becomes essential.

Existing agent benchmarks evaluate important aspects of this capability. Tool-calling benchmarks such as BFCL (Patil et al., 2025) focus on individual API invocations, including tool selection, schema following, and argument generation. More interactive benchmarks, including Tau$^2$-Bench (Barres et al., 2025), TripBench (Shen et al., 2026), VitaBench (He et al., 2025), SWE-bench (Jimenez et al., 2023), and Gaia2 (Froger et al., 2026), extend evaluation to multi-step tasks in domains such as travel planning, customer service, software engineering, mobile applications, and web workflows. However, many of these tasks are grounded in familiar domains with recurring solution templates: booking travel, applying customer-service policies, or editing code and running tests. Strong performance may therefore reflect learned domain conventions or short-range reactive tool use, rather than the ability to infer and execute a novel tool-use procedure in an unfamiliar environment.

Puzzle-based and escape-room-style benchmarks offer a complementary way to reduce reliance on familiar domain priors. Benchmarks such as EscapeBench (Qian et al., 2024) and PuzzleWorld (Li et al., 2025) probe exploratory reasoning in unfamiliar settings, but many operate in text-only action spaces or question-answering formats. As a result, they do not fully capture modern tool-grounded



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

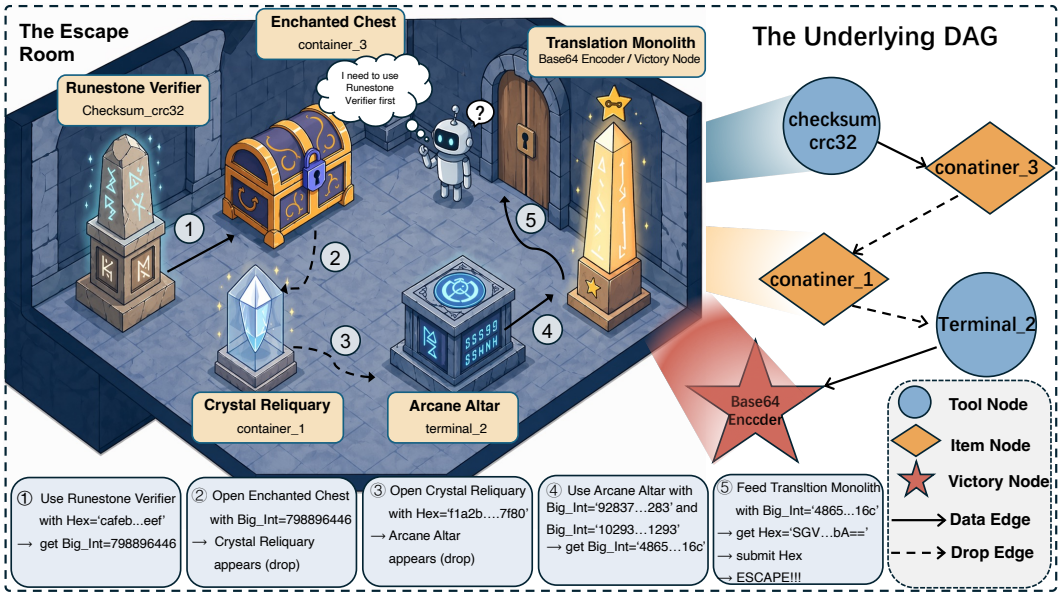



Figure 1: **Conceptual illustration of AgentEscapeBench.** The agent is placed in a themed escape room populated with unfamiliar tools and hidden items. It must explore the environment, invoke tools with correct parameters derived from narrative clues, and propagate intermediate outputs through a multi-step dependency chain to unlock the final exit.

deployments, where agents must call external functions with valid parameters, interpret structured feedback, and propagate outputs through executable dependency chains.

A further limitation is that long-horizon interaction does not necessarily require deep contextual dependency. In many benchmarks, tool calls can often be selected from the most recent observation, while information introduced much earlier is rarely an explicit prerequisite for later actions. This makes it difficult to diagnose whether failures stem from forgotten clues, incorrect state tracking, faulty propagation of intermediate results, or poor tool selection. A more diagnostic benchmark should therefore make long-context dependencies explicit, traceable, and automatically analysable. Table 4 summarises these distinctions.

To address these gaps, we introduce **AgentEscapeBench**, an escape-room-style framework for evaluating tool-grounded agent reasoning in unfamiliar environments; Figure 1 provides a conceptual overview. Each task is designed so that the agent cannot rely on domain-specific prior knowledge or a fixed workflow template. Instead, it must extract clues from the task description and incrementally revealed observations, decide which tools to call and how to parameterise them, propagate intermediate outputs through dependency chains, and revise its plan as new information becomes available. Individual tools use conventional computational semantics; the unfamiliarity lies in their task-level composition, implicit parameter provenance, and instance-specific workflow. AgentEscapeBench is therefore a controlled diagnostic of tool-grounded dependency reasoning rather than a simulation of every capability required in open-ended agent deployment. This design targets the underlying reasoning and tool-use capabilities needed for general-purpose agents, rather than surface-level mastery of a particular workflow.

We instantiate AgentEscapeBench with 270 tasks across five difficulty tiers, defined by DAGs with 5, 10, 15, 20, and 25 nodes. Each task is generated from compositional tool and item templates, supports incremental information disclosure, executes real external functions, and has a deterministic ground-truth answer for automated evaluation. We evaluate sixteen representative LLMs and conduct a human study under the same interaction and feedback conditions. Our results show that AgentEscapeBench differentiates models across a wide capability spectrum: performance drops sharply as interaction horizons lengthen and dependency threads become more interleaved, while trajectory-level analyses reveal model-specific failure signatures in behaviour, tool use, and problem-solving efficiency. Our main contributions are as follows:

- **Unfamiliar task-level tool composition.** We formulate AgentEscapeBench as an escape-room-style benchmark in which conventional tools are connected through instance-specific workflows and implicit parameter-source relationships, thereby targeting transfer across tool compositions rather than mastery of a recurring workflow.
- **Controllable DAG-based task generation.** We introduce an automated pipeline that composes tool and item templates into diverse DAG-structured tasks with incremental information disclosure, executable tool calls, and deterministic ground-truth answers.
- **Model and human evaluation across difficulty tiers.** We construct 270 instances across five controlled node-count tiers and evaluate sixteen LLMs alongside human participants, revealing how performance degrades as required interaction horizons and interleaved dependency complexity increase.
- **Trajectory- and step-level diagnostic analysis.** We analyse behavioural patterns, tool-calling errors, and problem-solving efficiency, exposing model-specific failure signatures that diagnose current capability bottlenecks and provide actionable signals for improving general-purpose agent capabilities.

## 2 Dataset Construction

AgentEscapeBench instances are produced by a six-stage automated pipeline (Figure 2). Each instance is a directed acyclic graph (DAG) of tool and item nodes, accompanied by natural-language narratives that serve as inferential clues. We provide a concise overview here; full algorithmic details appear in Appendix A.

**Stage 1: Template library.** We curate a library of **32 tool templates**, **2 terminal templates**, **16 item templates**, and **4 container templates**. Each tool template defines typed input ports, typed output ports, and a deterministic execution function. Tools span big-integer arithmetic, cryptographic primitives, encoding/decoding, file operations, and graph algorithms, among others (full list in Appendix A). Item and container templates carry information payloads and support an *incremental-disclosure mechanism*: when a node whose output ports include `ITEM` or `HIDDEN_ITEM` types is successfully resolved, the connected downstream nodes become newly visible to the agent, revealing items or tools that were not accessible at the outset.

**Stage 2: DAG skeleton generation.** Given a target node count $n$ (the difficulty level), we assemble a DAG skeleton using a **reverse-generation algorithm** (see Appendix A for full details). Starting from a randomly sampled goal node, the algorithm iteratively resolves pending input-port requirements by either reusing an existing node whose output type matches, or instantiating a fresh template. Structural constraints—acyclicity, single-use semantics for physical-object ports, and exclusivity for semantically sensitive ports—are enforced at every edge creation. Isomorphic duplicates are discarded. Node count is the controlled generation parameter rather than a synonym for critical-path depth. Higher tiers jointly increase the required interaction horizon and the number of interleaved dependency threads; Appendix D reports the complete structural analysis.

**Stages 3–4: Source annotation and value instantiation.** All leaf-level input ports (those with no incoming edge) are identified as sources requiring concrete seed values. An LLM generates these values, respecting type contracts and inter-port consistency constraints.

**Stage 5: Deterministic forward execution.** The DAG is executed in topological order: each tool reads its inputs, applies its deterministic function, and propagates results along outgoing edges. This forward pass is used only during construction: the agent never observes the complete DAG or a topological solution order, and must infer dependencies from local descriptions, incremental disclosure, and tool feedback. The terminal output constitutes the unique ground-truth flag. Instances that fail the forward pass are discarded.

**Stage 6: Narrative generation and validation.** An LLM generates background narratives for each node, partially disclosing the data-flow structure. Each instance is assigned one of eight thematic styles to promote diversity. A final automated quality check verifies that the instance is fully instantiated and solvable; only instances that pass are admitted to the benchmark.

**Human solvability validation.** Beyond automated checks, we conduct a human study to provide the strongest guarantee of data quality. All instances are distributed across multiple participants, with each instance attempted by at least one participant under the same interface and feedback conditions

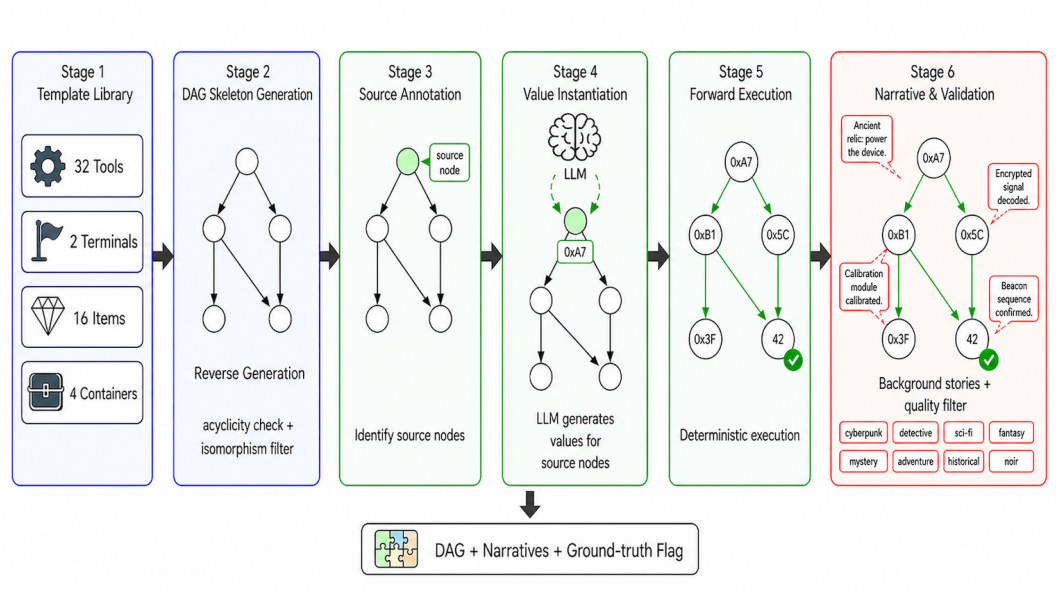



Figure 2: **Six-stage data construction pipeline of AgentEscapeBench.** Starting from a curated template library (Stage 1), a reverse-generation algorithm assembles a DAG skeleton (Stage 2), annotates source ports (Stage 3), instantiates concrete values via an LLM (Stage 4), executes the DAG forward to compute ground-truth outputs (Stage 5), and generates themed narratives with structural validation (Stage 6).

as the LLM agents. Every instance in the benchmark is successfully solved by at least one human participant, confirming that all automatically generated puzzles are well-formed and solvable.

# 3 Experiments

## 3.1 Evaluation Protocol

AgentEscapeBench contains 60 puzzle instances at each of difficulty levels 5, 10, 15, and 20, and 30 instances at difficulty 25, yielding 270 instances in total. Each evaluation episode proceeds as a multi-turn dialogue between the agent and a sandbox environment. At the start of an episode, the agent receives an initial observation that describes the room setting, the available tools and items, and the background narratives associated with visible nodes. On every subsequent turn, the agent may either *inspect* an entity to retrieve its detailed description, or *invoke* a tool by specifying the tool name and parameter values. The environment returns execution results for correct invocations or structured error feedback (e.g., wrong parameter value, missing dependency) for incorrect ones. When a successful tool invocation triggers a drop event, newly revealed items and tools become visible in the next observation, realising the incremental-disclosure mechanism described in Section 2. A complete solved trajectory illustrating this interaction loop is provided in Appendix E. An episode is scored as a *success* if and only if the agent produces the correct output value for the terminal (victory) node and submits it to the environment. The ground-truth value is deterministically computed during data construction (Section 2, Stage 5), ensuring unambiguous, fully automated evaluation with no human judgement required. We evaluate sixteen models in total; step budgets, model list, hyperparameters, and the complete system prompt are detailed in Appendix B. To obtain a human performance baseline, we randomly assign instances across multiple participants such that each instance is attempted exactly once by one participant, and report the aggregated success rate as the human baseline in Table 1.

## 3.2 Main Results

Table 1 reports three metrics across five difficulty levels: **success rate** (SR, the fraction of instances fully solved), **sub-problem resolution rate** (Sub., the average fraction of DAG nodes correctly

Table 1: **Main results on AgentEscapeBench.** We report success rate (**SR**, %), sub-problem resolution rate (**Sub.**, %), and hidden-node discovery rate (**Disc.**, %). **Bold**/underline: best/second-best model; $\Delta_{\text{SR}}$: SR drop from difficulty-5 to the hardest evaluated level (compare only shared endpoints); —: unmeasured.

| | Difficulty-5 | | | Difficulty-10 | | | Difficulty-15 | | | Difficulty-20 | | | Difficulty-25 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Model** | **SR** | **Sub.** | **Disc.** | **SR** | **Sub.** | **Disc.** | **SR** | **Sub.** | **Disc.** | **SR** | **Sub.** | **Disc.** | **SR** | **Sub.** | **Disc.** | $\Delta_{\text{SR}}$ |
| Human‡ | 98.3 | 96 | 100.0 | 95.0 | 94 | 97.3 | 85.0 | 92 | 96.8 | 81.7 | 90 | 95.9 | 80.0 | 86 | 95.4 | 18.3 |
| Claude-Opus-4.6 | 90.0 | 93 | 99.1 | **83.3** | 86 | 93.9 | 80.0 | 86 | **96.3** | **71.0** | **86** | **96.5** | **60.0** | **82** | **94.9** | 30.0 |
| Gemini-3.1-Pro-Preview | 91.7 | **94** | **100.0** | **83.3** | **89** | **94.5** | **83.3** | **89** | 94.6 | 60.0 | 82 | 86.6 | 13.3 | 74 | 74.7 | 78.4 |
| GPT-5.4 | **96.7** | 92 | 99.1 | **83.3** | 83 | 90.9 | 70.0 | 78 | 88.6 | 53.3 | 67 | 75.2 | 43.3 | 71 | 68.8 | 53.4 |
| Kimi-K2.5 | 95.0 | 92 | **100.0** | 81.7 | 86 | 91.1 | 70.0 | 78 | 81.5 | 31.7 | 66 | 80.1 | 26.7 | — | — | 68.3 |
| DeepSeek-Chat | 86.7 | 89 | 98.2 | 76.7 | 85 | 91.2 | 58.3 | 78 | 91.2 | 33.3 | 66 | 77.3 | — | — | — | 53.4 |
| DeepSeek-Reasoner | 78.3 | 89 | 99.1 | 65.0 | 81 | 89.9 | 58.3 | 77 | 85.8 | 36.7 | 68 | 80.4 | — | — | — | 41.6 |
| Gemini-3-Flash-Preview | 88.3 | 93 | **100.0** | 66.7 | 84 | 90.5 | 50.0 | 75 | 83.6 | 18.3 | 62 | 76.0 | — | — | — | 70.0 |
| Doubao-Seed-2.0-Pro | 86.7 | 89 | 98.5 | 63.3 | 76 | 85.8 | 31.7 | 59 | 70.9 | 15.0 | 49 | 70.9 | — | — | — | 71.7 |
| GPT-5 | 56.7 | 67 | 78.9 | 48.3 | 66 | 67.6 | 40.0 | 61 | 69.3 | 20.0 | 25 | 27.7 | — | — | — | 36.7 |
| MiniMax-M2 | 70.0 | 88 | 98.2 | 51.7 | 75 | 85.8 | 28.3 | 59 | 75.7 | 5.0 | 43 | 56.2 | — | — | — | 65.0 |
| Kimi-K2 | 51.7 | 55 | 62.1 | 35.0 | 48 | 53.3 | 23.3 | 43 | 55.0 | 11.7 | 32 | 47.8 | — | — | — | 40.0 |
| Qwen3-235B-A22B† | 88.3 | 89 | 99.1 | 58.3 | 75 | 87.0 | 16.7 | 50 | 63.9 | — | — | — | — | — | — | 71.6 |
| Qwen3-Next-80B-A3B† | 76.7 | 84 | 96.4 | 35.0 | 62 | 78.5 | 10.0 | 42 | 62.4 | — | — | — | — | — | — | 66.7 |
| Qwen3-32B† | 60.0 | 75 | 89.1 | 16.7 | 48 | 57.5 | 10.0 | 32 | 48.3 | — | — | — | — | — | — | 50.0 |
| Qwen3-14B† | 50.0 | 65 | 84.4 | 11.7 | 42 | 61.1 | 6.7 | 19 | 46.8 | — | — | — | — | — | — | 43.3 |
| Qwen3-8B† | 28.3 | 43 | 73.0 | 0.0 | 22 | 53.2 | 0.0 | 16 | 46.8 | — | — | — | — | — | — | 28.3 |

†Evaluated through difficulty-15. ‡Human participants use the same interface and feedback conditions; one attempt per instance.

resolved), and **hidden-node discovery rate** (Disc., the average fraction of initially hidden nodes uncovered). The upper model block contains eleven models evaluated on difficulty 5–20, while five open-source Qwen3 variants are evaluated on difficulty 5–15 (marked †). Stronger models are extended to higher difficulties; the human baseline, Claude-Opus-4.6, Gemini-3.1-Pro-Preview, GPT-5.4, and Kimi-K2.5 have success-rate measurements at all five tiers, and other comparisons use only shared tiers. Within each block, models are sorted by average success rate. Repeated runs for representative frontier and open-source models preserve the main trend; complete mean±standard-deviation results appear in Appendix D.4.

### 3.3 KEY OBSERVATIONS.

**Difficulty primarily scales the required interaction horizon.** Among the structural measures analysed in Appendix D.3, required interaction horizon is the strongest single predictor of success (standardized $\beta = -1.139$, SE $= 0.084$, AIC $= 3775$). Critical-path length is highly correlated with graph size ($\rho = 0.87$) and contributes no independent signal when entered jointly with horizon ($\beta = -0.01$, $p = 0.94$). The tiers therefore increase interaction scope and interleaved dependency complexity, rather than critical-path depth alone.

**Performance degradation across difficulty tiers.** All three metrics degrade monotonically with difficulty for every model, with the steepest drops occurring between difficulty-15 and difficulty-20, where most models lose more than half their difficulty-5 success rate. This degradation is not uniform: Gemini-3.1-Pro-Preview drops 78.4 percentage points from difficulty-5 to difficulty-25 and Doubao-Seed-2.0-Pro drops 71.7 points, while Claude-Opus-4.6 degrades most gracefully ($\Delta = 30.0$) and leads at all levels above 5. The divergence suggests that easy-task performance is an unreliable proxy for longer interaction horizons: strategies that suffice at difficulty-5 do not consistently scale as more sub-goals and dependencies must be coordinated. Difficulty-25 further separates the leading models: Claude-Opus-4.6 (60.0%) leads, followed by GPT-5.4 (43.3%), Kimi-K2.5 (26.7%), and Gemini-3.1-Pro-Preview (13.3%).

**Sub-problem resolution reveals the chaining bottleneck.** Sub-problem resolution and discovery rates reveal meaningful partial progress even when end-to-end success approaches zero. At difficulty-20, MiniMax-M2 solves only 5.0% of instances but still resolves 43% of sub-problems and discovers 56.2% of hidden nodes; Gemini-3.1-Pro-Preview maintains 82% sub-problem resolution and 86.6%

discovery even as its success rate drops to 60.0%. This pattern is consistent with a bottleneck in coordinating intermediate results across extended interactions: agents often resolve and discover many nodes without completing the full dependency structure.

**Reasoning models do not dominate.** Models with explicit chain-of-thought or reasoning capabilities do not uniformly outperform their non-reasoning counterparts. DeepSeek-Reasoner (avg. SR 59.6%) underperforms DeepSeek-Chat (avg. 63.8%) across all difficulty levels, suggesting that the bottleneck is not inference depth within one trace, but grounding reasoning in tool interactions and updating beliefs as new information arrives. Extended internal reasoning may introduce overconfidence or distraction when feedback must come from external calls rather than prior knowledge.

**Human–model gap reveals reasoning generalisation deficits.** Human success declines from 98.3% at difficulty-5 to 80.0% at difficulty-25 (18.3 points), whereas Claude-Opus-4.6 drops from 90.0% to 60.0% (30.0 points), and most models decline more steeply. At difficulty-15, humans achieve 85.0% while the best model reaches 83.3%; by difficulty-20, the gap widens to 81.7% versus 71.0%. Under the same interface and feedback conditions, the human results establish solvability and provide a performance reference, while the widening gap indicates that current models are less reliable at coordinating longer interaction horizons and interleaved dependencies.

# 4 Fine-Grained Trajectory Analysis

To move beyond aggregate success rates and diagnose *how* models succeed or fail, we perform a systematic offline analysis of every evaluation trajectory. For each of the 240 instances per model (60 per difficulty level 5–20), we parse the full interaction log and extract behavioural signals along three complementary dimensions: behavioural pattern analysis (§4.1), tool-calling error analysis (§4.2), and efficiency analysis (§4.3). The analysis covers eleven non-Qwen models across difficulty levels 5–20.

## 4.1 Behavioural Pattern Analysis

We compute three metrics that probe distinct capability dimensions: feedback-driven convergence, reasoning and state management, and environmental clue utilisation. Figure 3 shows the trend of each metric across difficulty levels.

**Feedback-driven convergence: source-node convergence speed.** The *source-node convergence speed* measures how many tool-call attempts are required before a source node (a leaf node with no data-flow predecessors) is correctly resolved. We focus on source nodes for two reasons. First, their parameter values are entirely derivable from information visible in the initial environment, removing any confound from upstream computation errors. Second, the evaluation environment provides explicit per-parameter feedback on each failed attempt—indicating which parameters are correct and which are not—giving the agent a concrete signal to narrow the search space. This metric therefore directly measures how effectively a model exploits structured feedback to converge on the correct parameter combination: a capable model should use each round of feedback to eliminate incorrect candidates and rapidly zero in on the golden values, while a less capable model will fail to leverage the same information and require many more attempts.

At difficulty-5, efficient models resolve source nodes in fewer than two attempts, whereas GPT-5 requires nearly five—a 3$\times$ gap despite receiving identical feedback. At difficulty-20, attempt counts rise across all models (range: 2.42–6.52). The increasing attempt counts at higher difficulties reflect the growing number of source nodes and the correspondingly larger candidate parameter space. The large spread across models—even when all receive identical per-parameter feedback—reveals that feedback-driven convergence is a genuine differentiating capability rather than a uniform bottleneck.

**Reasoning and state management: premature invocation rate.** The *premature invocation rate* measures the fraction of non-source node invocations that occur before all data-flow predecessors have been resolved—a direct indicator of out-of-order execution and faulty state tracking. A high rate means the agent attempts to use downstream tools before obtaining the required upstream outputs, a pattern that inevitably leads to incorrect parameter values and wasted steps. This metric probes a

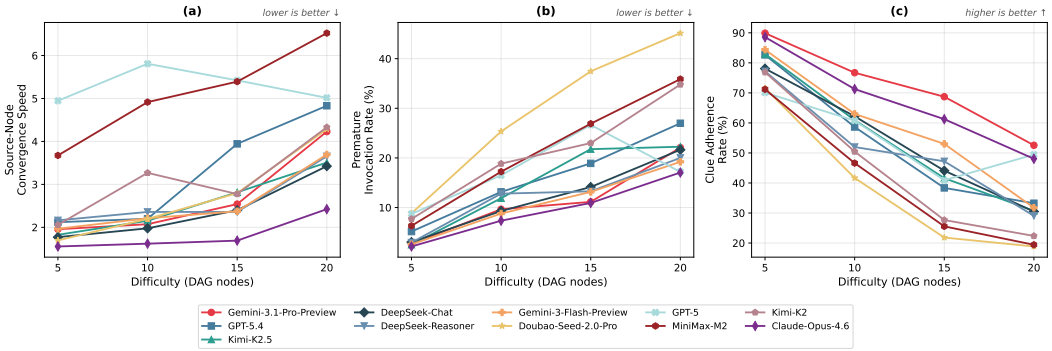



Figure 3: **Behavioural metric trends across difficulty levels.** (a) Source-node convergence speed (lower is better): the average number of attempts to correctly resolve a source node. (b) Premature invocation rate (lower is better): the fraction of non-source invocations made before all predecessors are resolved. (c) Clue adherence rate (higher is better): the fraction of downstream invocations whose arguments trace back to actual upstream outputs. All models degrade with difficulty, but at markedly different rates.

combination of reasoning ability (inferring the correct execution order from environmental clues) and working memory (maintaining an accurate record of which nodes have been solved and which remain pending as the episode progresses).

At difficulty-5, most models maintain rates below 9%, demonstrating competent topological ordering on shallow graphs. As difficulty grows, rates increase monotonically for every model, but with markedly different slopes. At difficulty-20, weaker models (Doubao-Seed-2.0-Pro: 45.1%; MiniMax-M2: 35.9%) invoke downstream nodes prematurely in over a third of their calls, while stronger models (Claude-Opus-4.6: 17.1%; Gemini-3.1-Pro-Preview: 22.1%) perform comparatively better, though still far from near-zero. The consistent degradation indicates that state management across extended interactions is a recurring bottleneck, with the measured behavioural signals worsening as interaction horizon and dependency interleaving increase.

**Environmental clue utilisation: clue adherence rate.** The *clue adherence rate* quantifies whether, when a model invokes a node with data-flow predecessors, the argument values it supplies can be traced back to actual outputs of those predecessors. Low adherence indicates that the model either substitutes self-generated or guessed values, or mistakenly uses outputs from non-dependent nodes, rather than correctly propagating the outputs of its actual upstream predecessors.

At difficulty-5, most models achieve 77–90% adherence, but by difficulty-20 a sharp separation emerges: Gemini-3.1-Pro-Preview (52.6%) and Claude-Opus-4.6 (48.0%) retain substantially higher adherence, while weaker models drop below 20%, essentially reverting to parametric guessing. This persistent advantage is the single strongest correlate of leading positions in the main evaluation, suggesting that the ability to faithfully propagate intermediate results through a multi-step tool chain is the core differentiating capability exposed by AgentEscapeBench.

### 4.2 Tool-Calling Error Analysis

We classify every failed tool invocation into a set of error categories by matching the structured feedback returned by the evaluation server (full taxonomy with definitions in Appendix B.2). Table 2 shows the distribution across all four difficulty levels.

Three error types reveal distinctive model-specific deficiencies:

- **Missing required parameter.** MiniMax-M2 (13.4 per instance at difficulty-20) and GPT-5 (8.7 at difficulty-10) exhibit dramatically high rates, suggesting systematic failure to enumerate the complete parameter set as tool schemas grow complex.

Table 2: **Error type distribution across difficulty levels** (average counts per instance). MiniMax-M2 and GPT-5 dominate the missing required parameter category at higher difficulties; GPT-5 uniquely exhibits wrong format errors. Error profiles are model-specific, pointing to distinct failure signatures.

| | Miss. Required Param. | | | | Wrong Node Type | | | | Wrong Call Format | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Model** | **D5** | **D10** | **D15** | **D20** | **D5** | **D10** | **D15** | **D20** | **D5** | **D10** | **D15** | **D20** |
| Claude-Opus-4.6 | 0.0 | 0.0 | 0.1 | 0.3 | 0.2 | 0.3 | 0.3 | 0.3 | 0.0 | 0.0 | 0.0 | 0.0 |
| Gemini-3.1-Pro-Preview | 0.1 | 0.1 | 0.1 | 0.5 | 0.0 | 0.0 | 0.0 | 0.1 | 0.0 | 0.0 | 0.0 | 0.0 |
| GPT-5.4 | 0.2 | 0.7 | 0.7 | 1.1 | 0.0 | 0.1 | 0.0 | 0.1 | 0.0 | 0.0 | 0.0 | 0.0 |
| Kimi-K2.5 | 0.1 | 1.2 | 0.2 | 6.2 | 0.1 | 0.2 | 0.2 | 0.5 | 0.0 | 0.0 | 0.0 | 0.0 |
| DeepSeek-Chat | 0.1 | 0.5 | 0.8 | 2.9 | 0.3 | 0.4 | 0.4 | 0.9 | 0.0 | 0.0 | 0.0 | 0.0 |
| DeepSeek-Reasoner | 0.2 | 0.5 | 0.5 | 1.8 | 0.5 | 0.5 | 0.7 | 0.9 | 0.0 | 0.0 | 0.0 | 0.0 |
| Gemini-3-Flash-Preview | 0.0 | 0.1 | 0.2 | 0.5 | 0.0 | 0.0 | 0.1 | 0.1 | 0.0 | 0.0 | 0.0 | 0.0 |
| Doubao-Seed-2.0-Pro | 0.1 | 0.5 | 0.2 | 1.3 | 0.1 | 0.3 | 0.3 | 0.4 | 0.0 | 0.0 | 0.0 | 0.0 |
| GPT-5 | 3.1 | 8.7 | 7.6 | 10.9 | 0.4 | 0.6 | 0.6 | 0.7 | 0.5 | 1.0 | 1.5 | 2.0 |
| MiniMax-M2 | 0.2 | 2.6 | 7.5 | 13.4 | 0.3 | 0.2 | 0.3 | 0.2 | 0.0 | 0.0 | 0.0 | 0.0 |
| Kimi-K2 | 0.1 | 0.8 | 0.8 | 1.9 | 0.1 | 0.1 | 0.2 | 0.1 | 0.0 | 0.0 | 0.0 | 0.0 |

Table 3: **Average tool calls per instance** across difficulty levels. Lower is better; **bold**/underline denotes fewest/second fewest. ⋆Min. Required is the theoretical lower bound.

| **Model** | **D5** | **D10** | **D15** | **D20** |
|---|---|---|---|---|
| Min. Required⋆ | 9.03 | 18.20 | 27.73 | 36.63 |
| Claude-Opus-4.6 | **13.1** | 36.0 | **58.7** | **101.2** |
| Gemini-3.1-Pro-Preview | 14.5 | **35.1** | 66.1 | 150.6 |
| GPT-5.4 | 13.2 | 52.2 | 120.0 | 213.6 |
| Kimi-K2.5 | 13.7 | 42.0 | 82.6 | 137.6 |
| DeepSeek-Chat | 15.6 | 39.4 | 78.6 | 130.9 |
| DeepSeek-Reasoner | 16.6 | 47.2 | 84.9 | 126.9 |
| Gemini-3-Flash-Preview | 14.6 | 46.7 | 87.6 | 152.2 |
| Doubao-Seed-2.0-Pro | 15.4 | 51.4 | 107.7 | 146.9 |
| GPT-5 | 24.8 | 61.5 | 108.4 | 120.4 |
| MiniMax-M2 | 20.3 | 58.4 | 116.1 | 151.2 |
| Kimi-K2 | 14.2 | 50.5 | 104.2 | 140.9 |

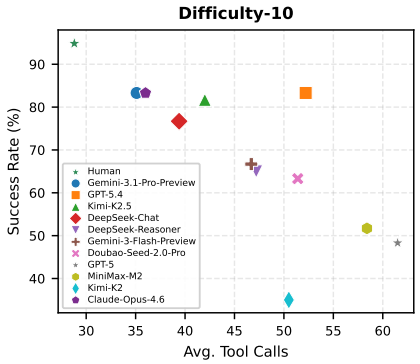



Figure 4: **Tool calls vs. success rate** at difficulty-10. Upper-left is better.

- **Wrong format.** GPT-5 uniquely suffers from this error across difficulty 5–15 while all other models produce near-zero counts, indicating a systematic format adherence failure for unfamiliar APIs.
- **Wrong node type.** DeepSeek-Reasoner and DeepSeek-Chat show the highest rates at difficulty-20, reflecting a failure to correctly identify which nodes are executable tools.

The distinct error profiles confirm that different models exhibit qualitatively different failure signatures, suggesting that targeted improvements could yield disproportionate gains.

### 4.3 Efficiency Analysis

Beyond correctness, we examine the *average number of tool calls* consumed per instance as a measure of problem-solving efficiency. Fewer tool calls indicate that the model navigates the dependency graph with less redundancy—fewer incorrect tool invocations and tighter execution plans. Table 3 reports the mean tool-call count for each model–difficulty pair.

From Table 3, efficiency differences *amplify* with difficulty. At difficulty-5, all models complete within 13–25 tool calls (a modest 1.9× spread); by difficulty-20, the range expands to 101–214 calls. Claude-Opus-4.6 and Gemini-3.1-Pro-Preview consistently consume the fewest calls across all levels, staying closest to the theoretical minimum, while GPT-5 and MiniMax-M2 require substantially more. Notably, GPT-5's low tool-call count at difficulty-20 (120.4, the fewest) does not reflect efficient problem-solving—its low success rate (20.0%) suggests it terminates episodes prematurely rather than persisting through the full dependency chain.

Figure 4 visualises the relationship between efficiency and success rate at difficulty-10, with the human baseline included for reference. The human data point (upper-left region) demonstrates

that humans solve the task with both high accuracy and few tool calls, representing the ideal of directed problem-solving. Among models, a clear positive correlation emerges: models in the upper-left quadrant (fewer calls, higher SR) such as Claude-Opus-4.6 and Gemini-3.1-Pro-Preview reach solutions via more direct execution paths, whereas models in the lower-right (more calls, lower SR) such as GPT-5 and MiniMax-M2 expend many calls unproductively. This correlation confirms that AgentEscapeBench rewards *directed* problem-solving rather than exhaustive search.

## 5 Related Work

Appendix Table 4 compares AgentEscapeBench with representative agent and puzzle-based benchmarks. Existing agent benchmarks often support real tool execution and automatic evaluation but operate in familiar domains with relatively fixed workflows; puzzle-based benchmarks stress unfamiliar exploration but often lack executable tools and fine-grained diagnostics. AgentEscapeBench bridges these directions through escape-room-style tasks, real tool execution, controlled node-count tiers with measured structural complexity, deterministic evaluation, and trajectory diagnostics.

**Agent Benchmarks with Domain-Specific Tasks.** A large body of agent benchmarks focuses on tasks drawn from well-defined, real-world domains where models carry substantial prior knowledge. Representative examples include software engineering (Jimenez et al., 2023), travel planning (Shen et al., 2026), retail and telecom services (Barres et al., 2025), smartphone applications (Froger et al., 2026), tool-call accuracy (Patil et al., 2025), code and web navigation (Liu et al., 2023), real-world applications (He et al., 2025), and long-horizon task completion (Luo et al., 2025). While these benchmarks measure important capabilities, their domain-specific nature means that a model can succeed largely through prior knowledge of the task workflow. AgentEscapeBench intentionally withholds a familiar task-level workflow: although individual tools retain conventional semantics, the agent must infer instance-specific parameter provenance and solution paths from the available evidence, targeting transfer across novel task compositions rather than mastery of one recurring workflow.

**Escape-Room and Puzzle-Based Benchmarks.** Several recent works share our insight that open-ended, unstructured tasks better probe exploratory reasoning. EscapeBench (Qian et al., 2024) adopted text-based room-escape scenarios; VisEscape (Lim et al., 2025) leveraged a visual room-escape framing; PuzzleWorld (Li et al., 2025) used multi-step puzzle-hunts; BigEscape (Tang & Sun, 2025) drew from television escape shows. However, all of these benchmarks either restrict the agent to predefined text actions or rely on pure question answering, without invoking real external APIs. AgentEscapeBench addresses this gap by coupling open-ended exploration with genuine function calling, making the evaluated capabilities directly transferable to real-world agentic deployments.

**Reasoning Benchmarks.** Mathematical reasoning benchmarks such as MATH (Hendrycks et al., 2021) and AIME (MAA, 2025) evaluate pure reasoning ability without tool interaction, while Zhang et al. (2023) studied tool-augmented math reasoning, demonstrating that tool use extends model competency into domains requiring deep abstract computation. More broadly, Qiao et al. (2024) showed that real tool use pushes past the capability boundaries of pure text-based inference. AgentEscapeBench complements these efforts by evaluating reasoning grounded in tool interactions: agents must reason about unfamiliar tools and adaptively chain their outputs rather than use tools to assist known problem types.

## 6 Conclusion

We presented **AgentEscapeBench**, a benchmark that evaluates general tool-grounded reasoning in deliberately unfamiliar problem settings. Tasks are constructed as directed acyclic graphs with no fixed solution pattern, requiring genuine function calls and adaptive planning under incremental information disclosure. Our evaluation of sixteen LLMs shows that the benchmark differentiates models across a wide performance spectrum, with gaps amplifying at higher-complexity tiers, while fine-grained trajectory analysis exposes model-specific failure signatures that can inform improvements in tool-grounded dependency reasoning.

### AI USE STATEMENT

We used LLM APIs during benchmark data synthesis to instantiate source values and generate task narratives, as described in Section 2 and Appendix A. All generated instances were validated through deterministic execution and automated consistency checks. The authors reviewed the resulting data and take full responsibility for the final content of this work.

### ETHICS STATEMENT

AgentEscapeBench consists of synthetic escape-room-style tasks and conventional computational tools. The benchmark does not contain sensitive personal data or target protected groups, and we do not identify foreseeable harmful applications or broader societal impacts specific to this work. The human evaluation was limited to solving synthetic benchmark instances through the same interface used for model evaluation and was used only to establish task solvability and provide a performance reference.

### REPRODUCIBILITY STATEMENT

We will release the complete benchmark dataset and all code required for task construction, evaluation, and analysis. The manuscript further provides the construction algorithm and validation procedure in Section 2 and Appendix A, the complete evaluation configuration and agent system prompt in Appendix B, structural and repeated-run analyses in Appendix D, and a complete example trajectory in Appendix E. Together with the planned code and data release, these materials are intended to support full reproduction of the reported results.

## REFERENCES

Anthropic. Claude Code: Agentic coding in the real world. `https://www.anthropic.com/`, 2025. Accessed: 2025.

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. $\tau^2$-bench: Evaluating conversational agents in a dual-control environment. *arXiv preprint arXiv:2506.07982*, 2025.

Romain Froger, Pierre Andrews, Matteo Bettini, Amar Budhiraja, Ricardo Silveira Cabral, Virginie Do, Emilien Garreau, Jean-Baptiste Gaya, Hugo Laurençon, Maxime Lecanu, Kunal Malkan, Dheeraj Mekala, Pierre Ménard, Gerard Moreno-Torres Bertran, Ulyana Piterbarg, Mikhail Plekhanov, Mathieu Rita, Andrey Rusakov, Vladislav Vorotilov, Mengjue Wang, Ian Yu, Amine Benhalloum, Grégoire Mialon, and Thomas Scialom. Gaia2: Benchmarking llm agents on dynamic and asynchronous environments. *arXiv preprint arXiv:2602.11964*, 2026.

Wei He, Yueqing Sun, Hongyan Hao, Xueyuan Hao, Zhikang Xia, Qi Gu, Chengcheng Han, Dengchang Zhao, Hui Su, Kefeng Zhang, et al. Vitabench: Benchmarking llm agents with versatile interactive tasks in real-world applications. *arXiv preprint arXiv:2509.26490*, 2025.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*, 2021.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

Hengzhi Li, Brendon Jiang, Alexander Naehu, Regan Song, Justin Zhang, Megan Tjandrasuwita, Chanakya Ekbote, Steven-Shine Chen, Adithya Balachandran, Wei Dai, et al. Puzzleworld: A benchmark for multimodal, open-ended reasoning in puzzlehunts. *arXiv preprint arXiv:2506.06211*, 2025.

Seungwon Lim, Sungwoong Kim, Jihwan Yu, Sungjae Lee, Jiwan Chung, and Youngjae Yu. Vis-escape: A benchmark for evaluating exploration-driven decision-making in virtual escape rooms. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 16031–16058, 2025.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. Agentbench: Evaluating llms as agents. *arXiv preprint arXiv:2308.03688*, 2023.

Haotian Luo, Huaisong Zhang, Xuelin Zhang, Haoyu Wang, Zeyu Qin, Wenjie Lu, Guozheng Ma, Haiying He, Yingsha Xie, Qiyang Zhou, et al. Ultrahorizon: Benchmarking agent capabilities in ultra long-horizon scenarios. *arXiv preprint arXiv:2509.21766*, 2025.

MAA. Aime 2025, 2025. URL `https://artofproblemsolving.com/wiki/index.php/AIME_Problems_and_Solutions`.

Shishir G Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E Gonzalez. The berkeley function calling leaderboard (bfcl): From tool use to agentic evaluation of large language models. In *Forty-second International Conference on Machine Learning*, 2025.

Cheng Qian, Peixuan Han, Qinyu Luo, Bingxiang He, Xiusi Chen, Yuji Zhang, Hongyi Du, Jiarui Yao, Xiaocheng Yang, Denghui Zhang, et al. Escapebench: Pushing language models to think outside the box. *arXiv e-prints*, pp. arXiv–2412, 2024.

Shuofei Qiao, Honghao Gui, Chengfei Lv, Qianghuai Jia, Huajun Chen, and Ningyu Zhang. Making language models better tool learners with execution feedback. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 3550–3568, 2024.

Yuanzhe Shen, Zisu Huang, Zhengyuan Wang, Muzhao Tian, Zhengkang Guo, Chenyang Zhang, Shuaiyu Zhou, Zengjie Hu, Dailin Li, Jingwen Xu, et al. Trip-bench: A benchmark for long-horizon interactive agents in real-world scenarios. *arXiv preprint arXiv:2602.01675*, 2026.

Zinan Tang and Qiyao Sun. Big escape benchmark: Evaluating human-like reasoning in language models via real-world escape room challenges. In *Proceedings of the Fourth Workshop on Generation, Evaluation and Metrics (GEM$^2$)*, pp. 488–503, 2025.

Zhenting Wang, Qi Chang, Hemani Patel, Shashank Biju, Cheng-En Wu, Quan Liu, Aolin Ding, Alireza Rezazadeh, Ankit Shah, Yujia Bao, et al. Mcp-bench: Benchmarking tool-using llm agents with complex real-world tasks via mcp servers. *arXiv preprint arXiv:2508.20453*, 2025.

Beichen Zhang, Kun Zhou, Xilin Wei, Xin Zhao, Jing Sha, Shijin Wang, and Ji-Rong Wen. Evaluating and improving tool-augmented computation-intensive math reasoning. *Advances in Neural Information Processing Systems*, 36:23570–23589, 2023.

# A Dataset Construction Details

This appendix provides full algorithmic details for the six-stage data construction pipeline summarised in Section 2.

## A.1 Stage 1: Tool and Item Template Library

The template library constitutes the atomic vocabulary from which every puzzle instance is assembled. It comprises four categories:

- **Tool templates (32).** Each template defines typed input ports, typed output ports, and a deterministic execution function. The library covers: big-integer arithmetic (multiplication, modular exponentiation, GCD, modular inverse), cryptographic primitives (SHA-256, MD5, HMAC-SHA256, AES-CBC decryption, RSA decryption), encoding/decoding (Base64, Hex, Zlib, ROT-N cipher, XOR), file and archive operations (ZIP extraction, CSV query, JSON path query, regex search), graph algorithms (shortest path), code execution (Python executor), numeric conversion (base converter, CRC32, Luhn check, IBAN validation), and text processing (Unicode normalisation, Bidi sanitisation, timezone conversion).
- **Terminal templates (2).** Special tool nodes that serve as the execution endpoint.
- **Item templates (16).** Information-carrying entities that may hold textual payloads, numerical values, or file references.
- **Container templates (4).** Entities with *drop mechanisms*: interacting with a container reveals hidden items or tools that were initially invisible.

Port types are drawn from a closed semantic vocabulary: `Big_Int`, `Text_Generic`, `Hex_String`, `Hex_String_Key_AES`, `Hex_String_IV_AES`, `File_Id`, `Code_Python`, `Item`, and `Hidden_Item`.

The tools were selected to require actual execution rather than reliable recovery by mental calculation or surface-form guessing, while retaining explicit type contracts and deterministic outcomes. Their individual semantics are conventional, but categories can be composed in instance-specific ways: for example, a decoded string may supply a file query, whose result may become an input to a cryptographic or numerical operation.

Each terminal template has two input ports. An invocation may populate either one port or both, and different correct input configurations return their corresponding outputs. An incorrect configuration returns an error rather than the terminal output.

Representative interface semantics include: big-integer tools map typed integer operands to exact arithmetic results and reject malformed or out-of-domain operands; cryptographic tools consume algorithm-specific values such as ciphertext, keys, or initialisation vectors and return the computed plaintext or digest, otherwise an execution error; encoders and decoders transform text, hexadecimal, or Base64 values and reject invalid encodings; file-query tools accept a file identifier plus a structured query such as a JSON path, CSV condition, or regular expression and return matching content; the shortest-path tool consumes a graph representation and endpoints and returns the computed path result; and the Python executor accepts `Code_Python` and returns its execution result or an error. These interfaces expose enough execution feedback to diagnose a local call without revealing the task's hidden dependency graph.

In addition, 68 *droppable tool variants* are automatically derived from the base tool templates by adding `ITEM`/`HIDDEN_ITEM` trigger ports. These variants are mixed into the generation pool with a configurable ratio (default 30%), enabling the benchmark to include tools that are only revealed through the incremental-disclosure mechanism.

## A.2 Stage 2: DAG Skeleton Generation (Reverse-Generation Algorithm)

Given a target node count $n$ (the difficulty level), the algorithm proceeds as follows:

**Algorithm 1** DAG Skeleton Generation via Reverse Growth

**Require:** Target node count $n$, template library $\mathcal{T}$
**Ensure:** DAG skeleton $G = (V, E)$

Select a random template $t \in \mathcal{T}$ that has $\geq 1$ input port
Create goal node $v_{\text{goal}}$ from $t$; set $V \leftarrow \{v_{\text{goal}}\}$, $E \leftarrow \emptyset$
Initialize pending queue $Q$ with all input ports of $v_{\text{goal}}$
**while** $|V| < n$ and $Q \neq \emptyset$ **do**
    Pop a random pending requirement $(v, p, \tau, r)$ from $Q$
    **if** $r > 5$ **then**
        **continue** ▷ skip unresolvable requirement
    **end if**
    Compute reuse probability $p_r \leftarrow 0.3 + 0.3 \cdot |V|/n$
    Find candidates $\mathcal{C} \leftarrow \{u \in V \mid u \text{ outputs type } \tau,\ u \neq v, \text{ no path } v \rightsquigarrow u\}$
    **if** $\mathcal{C} \neq \emptyset$ **and** $\text{rand}() < p_r$ **then**
        Pick $u \in \mathcal{C}$ at random; select available output port $q$ of type $\tau$
        Add edge $(u, q) \rightarrow (v, p)$ to $E$
    **else**
        Gather candidate templates from $\mathcal{T}$ with an output of type $\tau$
        **if** no candidates exist **then**
            Re-enqueue $(v, p, \tau, r{+}1)$ into $Q$; **continue**
        **end if**
        Sample template $t'$; create new node $u$ from $t'$
        $V \leftarrow V \cup \{u\}$; add edge $(u, q) \rightarrow (v, p)$ to $E$
        Enqueue all input ports of $u$ into $Q$ with $r = 0$
    **end if**
**end while**
**if** $G$ is isomorphic to a previously accepted skeleton **then**
    Discard $G$ and restart from line 1
**end if**
**return** $G$

1. **Sample final-goal node.** A template with at least one input port is selected uniformly at random. A node is instantiated from this template and designated as the win node; the puzzle's success condition is defined as producing the correct output of this node.
2. **Initialise pending queue.** All input ports of the final-goal node are enqueued as pending requirements, each represented as a tuple $(v, p, \tau, r)$: target node $v$, target port $p$, required type $\tau$, and retry count $r = 0$.
3. **Iterative resolution.** While |nodes| $< n$ and the queue is non-empty:
   (a) Pop a random pending requirement $(v, p, \tau, r)$; skip if $r > 5$.
   (b) With probability $0.3 + 0.3 \cdot (|\text{nodes}|/n)$, attempt to *reuse* an existing node whose output type matches $\tau$, subject to acyclicity and port-availability checks.
   (c) Otherwise, instantiate a *fresh* template node whose output matches $\tau$, add it to the graph, create the connecting edge, and enqueue all input ports of the new node.
4. **Structural constraints** enforced at every edge creation:
   - *Acyclicity*: BFS reachability check vetoes any edge that would close a cycle.
   - *Physical-object single-use*: ports of type `ITEM`/`HIDDEN_ITEM` may appear on at most one edge.
   - *Special-port exclusivity*: semantically sensitive input ports (regex patterns, JSON paths, OTP secrets, Base64 data) consume their source output exclusively—once connected, the source port is barred from further reuse.
   - *Same-target deduplication*: a target node's distinct input ports may not draw from the same source port.

**Isomorphism checking.** Each newly generated skeleton is checked against all previously accepted ones. Two DAGs are declared isomorphic if they share the same multiset of template identifiers, the

same multiset of edge patterns (source template, source port, target template, target port), and the same total node count. Isomorphic duplicates are discarded.

### A.3 Stage 3: Source Annotation

The annotation pass identifies every port that requires a concrete seed value before the forward execution pass. Two categories are flagged:

1. **Unconnected input ports**: any input port with no incoming edge is a leaf-level source.
2. **Drop-node outputs**: if a node's triggering input is of type `ITEM`/`HIDDEN_ITEM` (indicating activation by a physical object from a container), its output ports that feed downstream nodes are also flagged, because these values must be materialised before forward execution can propagate through them.

The output is a structured `source_init` map attached to the DAG metadata.

### A.4 Stage 4: Value Instantiation via LLM

For each port in the `source_init` map, an LLM (accessed via an OpenAI-compatible API) generates a concrete seed value. Each request provides the port's semantic type, its natural-language description, and any inter-port consistency constraints—for example:

- A modulus must exceed its base in a modular-exponentiation chain.
- A file path must be syntactically valid for the target operation.
- An AES key must be exactly 16/24/32 bytes in hex encoding.

This step introduces lexical and semantic diversity while ensuring every value satisfies its downstream type contract.

### A.5 Stage 5: Deterministic Forward Execution

Once all source ports carry concrete values, the DAG is executed in topological order by a deterministic simulator. Each tool node reads the values on its input ports, applies its pre-defined computation function, and writes results to its output ports. Values are immediately propagated along outgoing edges. Execution continues until every port holds a concrete value, yielding a fully instantiated DAG whose terminal output is the unique ground-truth flag.

A post-execution **completeness check** verifies that all non-physical-object ports have been assigned a value. Instances that fail this check are moved to a quarantine directory for post-hoc analysis.

### A.6 Stage 6: Narrative Generation and Quality Filtering

**Narrative generation.** Each node receives a background narrative generated by an LLM, conditioned on its DAG connectivity. The prompt supplies the node's type, its upstream and downstream relationships, and the thematic style. Narratives partially disclose the data-flow structure—hinting at which upstream output feeds a given downstream input—without revealing the complete solution path.

**Mapping the DAG to the agent environment.** The complete instantiated DAG remains internal to the generator and evaluator. The initial observation exposes only the opening narrative, currently visible entities, their node IDs and types, and a list of available parameter values whose source and intended use are not annotated. Investigating an entity reveals its local interface and background clue, but not the global graph. Hidden entities enter the visible set only after a successful drop-triggering interaction. For source nodes, the environment reports which submitted parameters are correct or incorrect without disclosing replacement values. For non-source nodes, it does not certify semantic parameter correctness: a call can execute with a type-valid but provenance-wrong upstream value, propagate an output, and lead to a later dead end. Consequently, agents must infer and, when needed, revise the latent dependency structure rather than receive a golden execution order.

To promote stylistic diversity and mitigate exploitation of surface-level textual cues, each instance is randomly assigned one of eight thematic styles: *cyberpunk*, *detective*, *sci-fi*, *fantasy*, *steampunk*, *post-apocalyptic*, *mystery*, and *techno-thriller*.

**Final validation.** Before admission to the benchmark pool, each instance passes automated checks:

- All required narrative fields are present.
- All node references are resolvable within the DAG.
- Every non-physical-object input port has a value; every non-physical-object output port has a value.
- The win condition is reachable via at least one complete tool-call sequence.

Instances that fail any check are moved to a rejection directory, guaranteeing that every accepted instance is well-formed, solvable, and has a provably correct unique answer.

# B Evaluation Details

**Step budget.** To accommodate the increasing number of tool calls required at higher difficulty levels, we set the maximum number of agent actions to 35, 80, 130, 160, and 200 for difficulty levels 5, 10, 15, 20, and 25, respectively. An episode terminates when the agent either submits the correct flag, exhausts its step budget, or explicitly declares failure.

**Model configuration.** All models are evaluated via their official API endpoints with a sampling temperature of 0.7. Each instance in the primary evaluation is evaluated in a single run; the main results do not average over multiple trials. Supplementary repeated runs for representative models quantify decoding variability on the fixed instances (Appendix D.4). We evaluate sixteen models spanning frontier closed-source systems (Claude-Opus-4.6, GPT-5.4, GPT-5, Gemini-3.1-Pro-Preview, Gemini-3-Flash-Preview, Kimi-K2.5, Kimi-K2, DeepSeek-Chat, DeepSeek-Reasoner, Doubao-Seed-2.0-Pro, MiniMax-M2) and open-source alternatives (Qwen3-235B-A22B, Qwen3-Next-80B-A3B, Qwen3-32B, Qwen3-14B, Qwen3-8B).

## B.1 Complete Agent System Prompt

The following is the complete instruction given to every evaluated agent (node IDs and environment-specific observations are supplied separately at episode initialisation). Line breaks and list formatting are adapted for typesetting; the instruction wording is otherwise unchanged.

---

You are an AI agent attempting to escape from an escape room.

Your goal is to find the victory string to escape the room. The system will tell you the name of the victory node (`win_node_id`) during initialization, which is your ultimate target node.

**Task Completion Strategy:**

**Identify the Victory Node:** The system will tell you which node is the victory node (`win_node_id`).

**Explore Dependency Paths:** The victory node usually has upstream dependency nodes. You need to:

- Investigate tools and items
- Understand the dependency relationships between them
- Call tools in the correct order to unlock them step by step

**Obtain the Victory String:** When you successfully call or investigate the victory node, you may get one or more output values from the returned result. You need to judge which value is the victory string.

**Autonomously Judge Submission Timing:** You need to judge when you have obtained the victory string based on the following clues:

- You have successfully called or investigated the victory node
- You have obtained the output result of that node

- The victory string could be any form of text (including text that may not intuitively seem like an "answer"—it could be code, numbers, seemingly unrelated text, etc.)

**Submit Answer:** Once you think you have obtained the victory string, use the `submit_answer` tool to submit it.

**Iterative Attempts:** If the submission fails, it means you did not submit the correct victory string and need to continue exploring.

**Note:**

- Do not try to call the victory node from the very beginning—it usually has upstream dependencies, and calling it directly will fail or not get the correct result.
- The victory string can be any form of text, including long paragraphs, descriptive text, code, numbers, etc.
- Only when you call `submit_answer` and get a success response (the system will explicitly tell you "Congratulations on successfully escaping the room! Your answer is correct: [your answer]"), is the task truly complete.
- If `submit_answer` returns an error, it means the answer you submitted is incorrect and you need to re-analyze.

**About the Narrative Description:**

- During initialization, you will see an opening narrative that sets the story background.
- When you investigate nodes, each will have its own Background Clue providing context and hints.
- The narrative descriptions use thematic names to make the story engaging.
- IMPORTANT: Always use the exact node ids from the "Entities in Environment" list for investigate and tool calls, not the thematic names from narratives.

**Operations you can perform:**

- **Call Tool Nodes:** After investigating a Tool type node, you can directly call it (using the node's ID as the function name), pass in parameters, and get output results.
- **Investigate Entity (`investigate`):** Investigate tools or items in the "Entities in Environment" list to view detailed information.
- **Use Item (`use_item`):** For some items that require parameters to "unlock" or "activate" (such as containers that need passwords, safes that need fingerprints, etc.), provide necessary parameters to use them.
- **Submit Answer (`submit_answer`):** When you believe you have obtained the victory string, submit it using this function to complete the task.

**About `use_item` Operation:**

- Only use `use_item` for Item type nodes that return a `required_inputs` field after investigation.
- Must use the exact parameter names shown in `required_inputs`. Format: `use_item(node_id="Item_ID",arguments={"param_name":"value"})`.

Tools and items in the environment each have a node ID, type, and description. Use investigate to view their detailed input/output interfaces. There are also some available parameters in the environment, which are just values without annotation of source or use.

**About "Background Clues":**

When you investigate tools or items, in addition to technical information (input/output parameters), you will also see "background clues." Background clues describe the role of each node and may explicitly indicate dependencies between nodes.

**About Parameter Validation Feedback:**

**Source Nodes (nodes with no upstream nodes):** When you use these nodes, the system will validate whether the parameters you used are correct.

- If all parameters are correct, the system will explicitly inform you "All parameters are correct!", indicating that you have successfully solved this node.
- If some parameters are incorrect, the system will tell you which parameters are wrong and which are correct, but will not tell you what the correct values are.
- The correct parameters for these nodes need to be inferred from the "Available Parameters" list.

**Non-Source Nodes (nodes with upstream nodes):** The system will NOT tell you whether the parameters you used are correct.

- Some correct inputs for these nodes come from outputs of upstream nodes (which will be hinted at in Background Clues).
- They may also require some initial parameters from the "Available Parameters" list.
- You need to reason about the correct parameters based on background clues and data flow relationships.

You need to:

- Understand the function of each tool (through description).
- Reason which available parameters are suitable as input for which tool.
- Reason the dependency relationship between tools (the output of one tool may be the input of another tool).
- Keep an open mind: Do not be limited by parameter names or "traditional usage" of tools. The output of one tool may be designed as an input parameter of another seemingly unrelated tool, even if it looks unconventional.
- Explore and try until you find the victory string.

**Important Reminders:**

- Read the tool descriptions and interface types carefully.
- Pay attention to input-output type matching (e.g., `Hex_String` needs hexadecimal strings).
- Data flow tracking is crucial: When a tool successfully executes, its output value may be exactly the parameter needed by another tool or item. Don't ignore the return values of tool calls, they are often key links in the puzzle chain.
- Cross-step reasoning: The puzzle process may require multiple steps to be linked. For example: Tool A's output → Tool B's input → Tool B's output → Item C's parameter. Build awareness of this multi-step data flow.
- Do not try to investigate names inferred from the following sources:
  - Descriptive elements in the narrative description (such as "log", "screen", etc.).
  - Data values returned by tools (such as node names in paths).
  - Input and output parameter names of tools (such as `text_output`, `hex_output`).
  - "Possibly existing" items inferred based on common sense (such as `note_1`, `hint_*`).
- Don't be discouraged if tool calls fail, try other methods.
- Track operations you have already tried to avoid repetition.

---

### B.2 Error Type Taxonomy

We classify every failed tool invocation (excluding `investigate` and `submit_answer` operations) into the following categories based on the structured feedback returned by the evaluation server.

1. **Missing required parameter** — The agent omits one or more mandatory input parameters when invoking a tool or item. Triggered when the server reports missing parameters, an empty parameter set, or a partial parameter submission.
2. **Wrong node type** — The agent attempts an operation incompatible with the node's type, such as calling `use_item` with arguments on a simple Item that should be accessed via `investigate`.
3. **Repeated solved node** — The agent attempts to unlock an item that has already been successfully solved in a previous step, wasting a turn.
4. **Wrong parameter type** — The agent provides a parameter value with an incorrect data type (e.g., passing an integer where a hex string is expected).
5. **Node not visible** — The agent attempts to invoke a node that does not currently exist in the environment, or mistakenly treats a non-node entity as an invocable node.
6. **Node not exist** — The agent references a node ID that does not exist in the current scenario, typically due to hallucinating a node name or confusing template names with instance-specific IDs.
7. **Wrong format** — The agent's tool-call request is structurally malformed, missing essential fields such as `node_id`.
8. **Other** — Errors that do not match any of the above classification rules.

Table 4: Comparison of representative *agent benchmarks* and *puzzle-based benchmarks*. The two panels together reproduce the complete comparison: each trait is fully addressed (✓), partially addressed (△), or not addressed (✗).

**Panel A: Generalization and tool-grounded dependency reasoning**

| Benchmark | OOD Setting | No Fixed Workflow | Real Tool Execution | Deep Contextual Dependency | Minimal Prior Knowledge |
|---|---|---|---|---|---|
| BFCL (Patil et al., 2025) | ✗ | △ | ✓ | ✗ | ✗ |
| $\tau^2$-Bench (Barres et al., 2025) | ✗ | △ | ✓ | ✗ | △ |
| TripBench (Shen et al., 2026) | ✗ | △ | ✓ | △ | △ |
| VitaBench (He et al., 2025) | ✗ | △ | ✓ | △ | △ |
| SWE-bench (Jimenez et al., 2023) | ✗ | △ | ✓ | ✗ | ✗ |
| EscapeBench (Qian et al., 2024) | ✓ | ✓ | ✗ | △ | △ |
| PuzzleWorld (Li et al., 2025) | ✓ | ✓ | ✗ | △ | △ |
| MCP-Bench (Wang et al., 2025) | △ | △ | ✓ | ✗ | ✗ |
| **AgentEscapeBench (ours)** | ✓ | ✓ | ✓ | ✓ | ✓ |

**Panel B: Evaluation, diagnostics, and scalability**

| Benchmark | Deterministic Auto Evaluation | Step-level Diagnostics | Human Eval. & Comparison | Controlled Difficulty |
|---|---|---|---|---|
| BFCL (Patil et al., 2025) | ✓ | ✗ | ✗ | △ |
| $\tau^2$-Bench (Barres et al., 2025) | ✓ | △ | ✗ | ✗ |
| TripBench (Shen et al., 2026) | ✓ | △ | ✗ | ✓ |
| VitaBench (He et al., 2025) | ✗ | △ | ✗ | △ |
| SWE-bench (Jimenez et al., 2023) | ✓ | ✗ | ✗ | ✗ |
| EscapeBench (Qian et al., 2024) | △ | ✗ | ✗ | △ |
| PuzzleWorld (Li et al., 2025) | △ | ✗ | ✗ | △ |
| MCP-Bench (Wang et al., 2025) | ✗ | ✓ | ✗ | ✗ |
| **AgentEscapeBench (ours)** | ✓ | ✓ | ✓ | ✓ |

Table 5: Graph structure by difficulty tier. Values after ± are standard deviations.

| Tier | $n$ | Critical path | Range | Width | Branching | Depth/size |
|---|---|---|---|---|---|---|
| 5 | 60 | $4.47 \pm 0.68$ | 3–5 | $1.48 \pm 0.79$ | 1.02 | 0.89 |
| 10 | 60 | $7.03 \pm 1.28$ | 5–10 | $2.92 \pm 0.85$ | 1.18 | 0.70 |
| 15 | 60 | $9.63 \pm 1.98$ | 5–15 | $3.95 \pm 1.15$ | 1.26 | 0.64 |
| 20 | 60 | $11.88 \pm 2.50$ | 8–17 | $5.32 \pm 1.42$ | 1.26 | 0.59 |
| 25 | 30 | $13.20 \pm 2.85$ | 7–18 | $6.77 \pm 1.72$ | 1.30 | 0.53 |

# C Supplementary Benchmark Comparison

# D Structural and Repeated-Run Analyses

## D.1 Graph Structure Across Difficulty Tiers

Node count is the controlled generation parameter, but it does not uniquely determine critical-path length, width, or branching. Table 5 reports these complementary properties. Critical path is the number of nodes on the longest directed path; width is the maximum antichain size; branching is mean out-degree over non-terminal nodes; and depth/size is critical-path length divided by node count. As the node budget grows, both depth and width increase, while depth/size decreases, showing that larger tasks add parallel dependency threads as well as longer paths.

We additionally compute topological measures exactly by dynamic programming over each graph's order ideals. The *concurrent working set* is, minimised over valid solving orders, the largest number of already resolved nodes whose outputs are still required. A chain has a working set of one regardless of its length, so this measure increases specifically when dependency threads interleave. *Live branching* measures the number of currently executable alternatives, and *order-comparable pairs* is the fraction

Table 6: Topological complexity by difficulty tier.

| Tier | Concurrent working set | Valid solving orders | Live branching | Order-comparable pairs |
|---|---|---|---|---|
| 5 | 1.47 | $10^{0.2}$ | 1.16 | 0.90 |
| 10 | 2.68 | $10^{1.7}$ | 1.73 | 0.74 |
| 15 | 3.75 | $10^{3.6}$ | 2.14 | 0.70 |
| 20 | 4.12 | $10^{6.2}$ | 2.70 | 0.66 |
| 25 | 5.00 | $10^{9.4}$ | 3.24 | 0.62 |

Table 7: Minimum and realised interaction horizons. Realised interactions and ratios use successful runs only.

| Tier | Minimum | Per node | Realised | Ratio |
|---|---|---|---|---|
| 5 | 9.03 | 1.81 | 11.89 | $1.32\times$ |
| 10 | 18.20 | 1.82 | 29.33 | $1.61\times$ |
| 15 | 27.73 | 1.85 | 55.73 | $2.01\times$ |
| 20 | 36.63 | 1.83 | 82.17 | $2.24\times$ |

Table 8: Single-predictor logistic analyses of task success. Coefficients are standardized per one standard deviation; lower AIC indicates better fit.

| Predictor | $\beta$ per SD | SE | AIC |
|---|---|---|---|
| Required interaction horizon | $-1.139$ | 0.084 | 3775 |
| Node count | $-1.069$ | 0.090 | 3845 |
| Valid solving orders | $-0.932$ | 0.086 | 3985 |
| Concurrent working set | $-0.885$ | 0.092 | 4023 |
| Graph width | $-0.865$ | 0.084 | 4041 |
| Critical-path length | $-0.828$ | 0.093 | 4077 |

of node pairs whose order is fixed by the DAG. Table 6 shows that the working set more than triples and the space of valid solving orders grows from approximately $10^{0.2}$ to $10^{9.4}$ across tiers.

### D.2 Required Interaction Horizon

The minimum required interaction horizon counts the environment interactions needed to complete a task along an optimal valid execution. It scales at approximately 1.8 interactions per node (Table 7). The realised values are computed only over successful runs so that they are not inflated by trajectories that merely exhaust the step budget. The increasing realised-to-minimum ratio indicates that larger tasks impose progressively more exploratory or corrective work beyond their theoretical minimum. We do not report a realised difficulty-25 value because the corresponding aggregate was not computed.

### D.3 Structural Predictors of Success

We fit logistic models with model fixed effects and standard errors clustered by task, because all models encounter the same instances but differ in baseline ability. Table 8 ranks single structural predictors by AIC. Required interaction horizon is the strongest predictor. Critical-path length is also negatively associated with success when used alone, but it is highly correlated with graph size ($\rho = 0.87$). When critical-path length and required interaction horizon are entered jointly, the standardized critical-path coefficient is $-0.01$ ($p = 0.94$), providing no independent signal beyond horizon.

### D.4 Repeated-Run Evaluation

The main table reports one evaluation run per model–instance pair. To quantify variability induced by stochastic decoding at temperature 0.7, we repeat the evaluation for representative frontier and

Table 9: Repeated-run results for GPT-5.4 and Kimi-K2.5 (mean ± standard deviation across runs; all metrics in %).

| Model | Tier | Runs | SR | Sub. | Disc. |
|---|---|---|---|---|---|
| GPT-5.4 | 5 | 4 | $97.5 \pm 1.7$ | $92.4 \pm 0.5$ | $99.3 \pm 1.4$ |
| | 10 | 4 | $87.5 \pm 2.9$ | $85.0 \pm 1.5$ | $91.2 \pm 0.6$ |
| | 15 | 4 | $67.9 \pm 2.5$ | $76.0 \pm 1.6$ | $84.3 \pm 2.7$ |
| | 20 | 4 | $64.2 \pm 4.0$ | $70.7 \pm 0.7$ | $79.6 \pm 2.4$ |
| | 25 | 4 | $43.3 \pm 4.7$ | $67.6 \pm 1.9$ | $79.8 \pm 1.6$ |
| Kimi-K2.5 | 5 | 8 | $91.3 \pm 2.3$ | $90.5 \pm 1.3$ | $97.1 \pm 2.2$ |
| | 10 | 8 | $68.1 \pm 4.1$ | $80.4 \pm 2.9$ | $85.4 \pm 4.0$ |
| | 15 | 8 | $47.9 \pm 3.1$ | $71.0 \pm 2.5$ | $76.9 \pm 3.8$ |
| | 20 | 8 | $29.0 \pm 5.3$ | $64.6 \pm 1.3$ | $73.6 \pm 3.5$ |

Table 10: Repeated-run results for Qwen models (mean ± standard deviation across runs; all metrics in %).

| Model | Tier | Runs | SR | Sub. | Disc. |
|---|---|---|---|---|---|
| Qwen3-14B | 5 | 6 | $67.2 \pm 5.9$ | $75.5 \pm 2.2$ | $85.7 \pm 3.2$ |
| | 10 | 6 | $30.8 \pm 4.4$ | $52.5 \pm 0.8$ | $68.1 \pm 1.7$ |
| | 15 | 6 | $7.2 \pm 1.7$ | $28.2 \pm 0.7$ | $46.0 \pm 2.1$ |
| Qwen3-235B-A22B-Instruct-2507 | 5 | 8 | $47.1 \pm 4.0$ | $64.2 \pm 2.1$ | $68.6 \pm 2.3$ |
| | 10 | 8 | $26.7 \pm 2.4$ | $58.3 \pm 1.6$ | $60.5 \pm 2.6$ |
| | 15 | 8 | $11.5 \pm 3.9$ | $45.0 \pm 2.5$ | $49.9 \pm 2.4$ |
| Qwen3-32B | 5 | 6 | $66.7 \pm 7.3$ | $78.5 \pm 3.2$ | $88.4 \pm 1.7$ |
| | 10 | 6 | $36.1 \pm 6.6$ | $60.4 \pm 3.5$ | $70.4 \pm 5.1$ |
| | 15 | 6 | $10.6 \pm 3.6$ | $35.9 \pm 2.3$ | $52.8 \pm 3.1$ |
| Qwen3-8B | 5 | 6 | $39.2 \pm 1.4$ | $60.3 \pm 2.1$ | $72.8 \pm 3.4$ |
| | 10 | 6 | $11.4 \pm 3.9$ | $38.5 \pm 2.2$ | $49.2 \pm 3.3$ |
| | 15 | 6 | $4.7 \pm 1.3$ | $19.1 \pm 0.9$ | $33.6 \pm 3.8$ |
| Qwen3-Next-80B-A3B-Instruct | 5 | 8 | $67.1 \pm 3.4$ | $78.0 \pm 1.9$ | $85.4 \pm 1.7$ |
| | 10 | 8 | $31.5 \pm 3.8$ | $58.5 \pm 1.6$ | $66.0 \pm 2.2$ |
| | 15 | 8 | $12.5 \pm 2.8$ | $40.9 \pm 1.3$ | $49.9 \pm 1.7$ |

open-source models. Tables 9 and 10 report means and standard deviations across runs. The principal decline across tiers persists under repetition. These standard deviations measure repeated-decoding variability on the fixed benchmark instances; they do not quantify uncertainty from drawing a new finite task sample from the generator.

# E EXAMPLE SOLVING TRAJECTORY

We present a complete interaction trace of Claude-Opus-4.6 solving a difficulty-10 instance. The agent successfully solves the puzzle in 12 turns (3 investigation rounds comprising 10 investigations, 8 tool-call rounds comprising 10 tool invocations, and 1 answer submission) with zero errors. Figures 5–7 show the full context visible to the model, including the system prompt, environment initialization, and all agent–environment interactions.

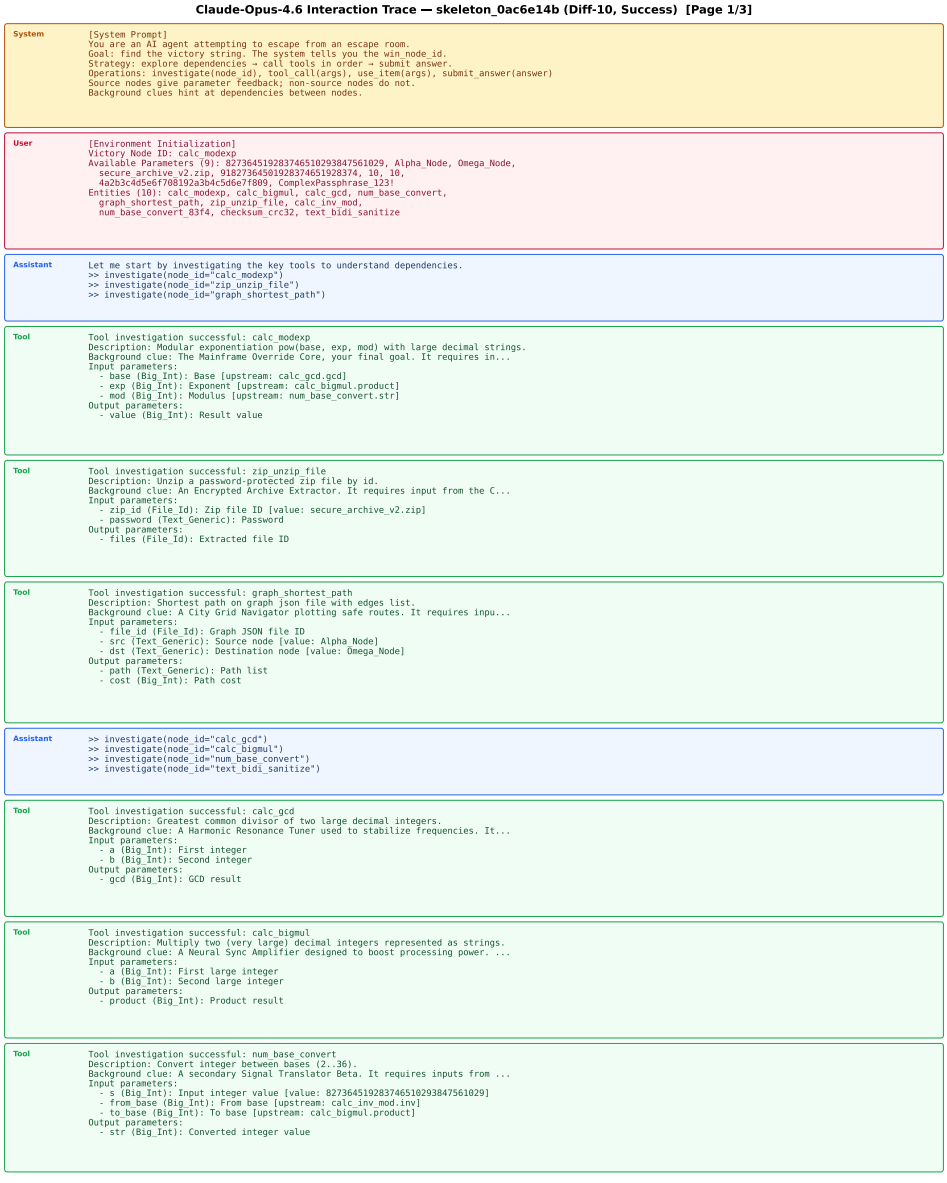



Figure 5: Interaction trace (page 1/3): system prompt, environment initialization, and first investigation round.

**Claude-Opus-4.6 Interaction Trace — skeleton_0ac6e14b (Diff-10, Success) [Page 2/3]**

**Tool**
```
Tool investigation successful: text_bidi_sanitize
Description: Strip bidi control chars.
Background clue: A Corrupted Data Purger module built into the side of a rust...
Input parameters:
  - text (Text_Generic): Text
Output parameters:
  - text (Text_Generic): Sanitized text
```

**Assistant**
```
>> investigate(node_id="calc_inv_mod")
>> investigate(node_id="num_base_convert_83f4")
>> investigate(node_id="checksum_crc32")
```

**Tool**
```
Tool investigation successful: calc_inv_mod
Description: Modular inverse of a modulo mod (large decimal strings).
Background clue: A volatile Quantum Inverter matrix. It requires inputs from ...
Input parameters:
  - a (Big_Int): Integer a [upstream: num_base_convert_83f4.str]
  - mod (Big_Int): Modulus [upstream: calc_bigmul.product]
Output parameters:
  - inv (Big_Int): Modular inverse
```

**Tool**
```
Tool investigation successful: num_base_convert_83f4
Description: Convert integer between bases (2..36).
Background clue: A Signal Translator Alpha unit, heavily modified with illega...
Input parameters:
  - s (Big_Int): Input integer value
  - from_base (Big_Int): From base
  - to_base (Big_Int): To base
Output parameters:
  - str (Big_Int): Converted integer value
```

**Tool**
```
Tool investigation successful: checksum_crc32
Description: CRC32 over hex bytes.
Background clue: An Integrity Scanner pulsing with a faint blue light. It ver...
Input parameters:
  - hex (Hex_String): Hex data
Output parameters:
  - hex (Hex_String): CRC32 Hex
  - int (Big_Int): CRC32 Int
```

**Assistant**
```
Now I can trace the dependency chain. Let me work through this:

1. **text_bidi_sanitize** → password for zip (input: `ComplexPassphrase_123!`)
2. **zip_unzip_file** → file_id for graph
3. **graph_shortest_path** → cost (used in calc_gcd and calc_big
[...reasoning...]
>> text_bidi_sanitize(text="ComplexPassphrase_123!")
>> checksum_crc32(hex="4a2b3c4d5e6f708192a3b4...")
>> num_base_convert_83f4(from_base="10", s="91827364501928374651928374...", to_base="10")
```

**Tool**
```
Tool text_bidi_sanitize invoked
Output: {"text": "ComplexPassphrase_123!"}
All parameters correct!
```

**Tool**
```
Tool checksum_crc32 invoked
Output: {"hex": "3ba77e11", "int": 1000832529}
All parameters correct!
```

**Tool**
```
Tool num_base_convert_83f4 invoked
Output: {"str": "91827364501928374651928374"}
All parameters correct!
```

**Assistant**
```
>> zip_unzip_file(password="ComplexPassphrase_123!", zip_id="secure_archive_v2.zip")
```

**Tool**
```
Tool zip_unzip_file invoked
Output: {"files": "topology_graph.json"}
```

**Assistant**
```
>> graph_shortest_path(dst="Omega_Node", file_id="topology_graph.json", src="Alpha_Node")
```

**Tool**
```
Tool graph_shortest_path invoked
Output: {"path": "['Alpha_Node', 'Router_X', 'Gateway_7', 'Switch_9', 'Omega_Node']",
"cost": "48291"}
```

**Assistant**
```
Now calc_bigmul needs graph cost (48291) and checksum int (1000832529):
>> calc_bigmul(a="48291", b="1000832529")
```

**Tool**
```
Tool calc_bigmul invoked
Output: {"product": "48331203657939"}
```

**Assistant**
```
Now calc_gcd (graph cost + Signal Translator Alpha output):
>> calc_gcd(a="48291", b="91827364501928374651928...")
```

**Tool**
```
Tool calc_gcd invoked
Output: {"gcd": "1"}
```

Figure 6: Interaction trace (page 2/3): remaining investigations, dependency reasoning, and tool execution chain.

**Claude-Opus-4.6 Interaction Trace — skeleton_0ac6e14b (Diff-10, Success) [Page 3/3]**

**Assistant**

```
Now calc_inv_mod (a=Signal Translator Alpha output, mod=bigmul product):
>> calc_inv_mod(a="91827364501928374651 92...", mod="48331203657939")
```

**Tool**

```
Tool calc_inv_mod invoked
Output: {"inv": "22096807214944"}
```

**Assistant**

```
Now num_base_convert (s=827364519283746510293847561029, from_base=inv, to_base=product).
But bases must be 2-36... these values are huge. Let me re-think.

Wait, the from_base and to_base need to be 2-36. The upstream values are massive. Maybe I
misi
[...reasoning...]
>> num_base_convert(from_base="22096807214944", s="827364519283746510 2938...",
to_base="48331203657939")
```

**Tool**

```
Tool num_base_convert invoked
Output: {"str": "827364519283746510293847561029"}
```

**Assistant**

```
OK, so the output is the same as input. Now calc_modexp:
>> calc_modexp(base="1", exp="48331203657939", mod="827364519283746510 2938...")
```

**Tool**

```
Tool calc_modexp invoked
Output: {"value": "1"}
```

**Assistant**

```
Result is 1. Let me try submitting:
>> submit_answer(answer="1")
```

**Tool**

```
Congratulations on successfully escaping the room! Your answer is correct: 1
```

Figure 7: Interaction trace (page 3/3): final computation steps and successful answer submission.

# F LIMITATIONS

**Benchmark scope.** AgentEscapeBench is designed as a controlled diagnostic of tool-grounded dependency reasoning. Its deterministic synthetic environments enable precise control over task structure and fine-grained trajectory analysis, while complementary evaluation in open-ended, multilingual, and dynamically changing environments remains a useful direction for future work.

**Evaluation coverage.** The primary evaluation covers sixteen models and 270 fixed benchmark instances, with a common step budget at each difficulty tier to ensure comparable conditions. Supplementary repeated runs for representative frontier and open-source models (Appendix D.4) show that the main performance trend persists under decoding variability. Future evaluations can extend this analysis to additional generated instances, models, and interaction-budget settings.

**Task composition.** Individual tools use conventional computational semantics so that performance is not confounded by undocumented tool behavior. The benchmark instead concentrates difficulty in instance-specific composition, implicit parameter provenance, incremental disclosure, and interleaved dependencies. Accordingly, the reported results should be interpreted as measurements of these targeted capabilities rather than as a comprehensive ranking of every capability needed for general-purpose agent deployment.

**Human baseline.** The human study establishes the solvability of accepted instances and provides a performance reference under the same interface and feedback conditions. Its role is to calibrate task feasibility and difficulty rather than to characterize all aspects of human problem solving.