CRAB-Bench: Evaluating LLM Agents under Complex Task Dependencies and Human-aligned User Simulation

Anonymous ACL submission

Abstract

Evaluating LLM agents in realistic service scenarios requires complex task dependencies, imperfect user behavior, and an evaluation that accommodates multiple valid solutions. We introduce CRAB-Bench (Constraint-based Realistic Agent Benchmark) and RUSE (Realistic User Simulation Engine) to address this gap. CRAB-Bench generates tasks via a constraint graph over multiple interdependent entities with structured distractors, requiring agents to reason carefully over thousands of misleading candidates where only a tiny fraction of solutions are valid. RUSE replaces cooperative, template-like simulators with realistic users grounded in human behavioral studies, instantiated across diverse personas and four behavioral dimensions. Experiments on four frontier LLM agents show that the best model achieves only 61% pass@1 on CRAB-Bench, and switching to RUSE causes further drops of up to 57%, concentrated in task-solving ability rather than conversational quality. Information Disclosure is the most damaging behavioral dimension, and agents interacting with RUSE are less likely to admit mistakes, instead masking errors through implicit corrections.

1 Introduction

LLM-based agents are increasingly deployed in complex, multi-turn service scenarios such as travel planning, customer support, and technical assistance (Chiang et al.; Zhang et al., 2025; Yao et al.; Barres et al.). Systematically evaluating agents in these settings is challenging for three reasons. First, real user requests involve *multi-step task dependencies*: satisfying one subtask implicitly constrains others, requiring the agent to plan across entities and propagate consequences without being explicitly told to do so. Second, real users are *imperfect*: they disclose information incrementally, communicate ambiguously, and react emotionally to errors (Laban et al., 2026; Qian et al., 2025; Zhou et al.). Third, most tasks admit *multiple valid solutions*, so evaluation cannot simply compare agent output against a single ground-truth record.

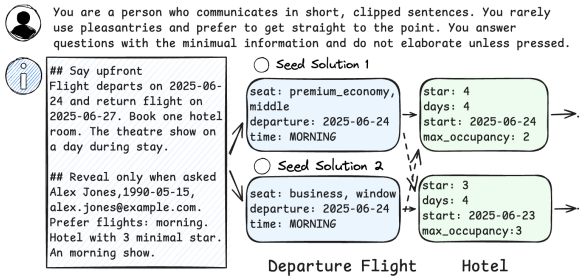

Figure 1: One task in CRAB-Bench with user persona and information control. 16 solutions satisfy the user requirements due to the combinations of different parts of seed solutions (as shown in the dotted lines).

Existing benchmarks address these challenges only partially. Single-tool and single-interaction benchmarks (Qin et al.; Guo et al., 2024; Liu et al.; Jimenez et al.; Vero et al., 2025) omit inter-task dependencies and multi-turn dynamics. Multi-turn benchmarks such as τ-bench (Yao et al.) and τ^2-bench (Barres et al.) use cooperative, template-like user simulators and rigid ground-truth evaluation, and have already been largely saturated. Meanwhile, recent studies show that LLM-based user simulators systematically miscalibrate evaluation and diverge substantially from real human behavior (Seshadri et al.; Zhou et al.), raising doubts about the gaps between benchmark performance and realistic user scenarios.

To address these gaps, we propose CRAB-Bench (Constraint-based Realistic Agent Benchmark), an agentic evaluation framework with 3 components.

- **Constraint graph-based task generation.** We model each task as a constraint graph over subtask nodes with domain constraints and edge constraints to capture real-world coherence requirements (e.g., a hotel check-in must follow a flight's arrival). We propose a constraint

graph-based generation pipeline to automatically generate seed solutions via CSP (Constraint Satisfaction Problem) solver and populate the database with distractors, ensuring both solvability and complexity.

- **Human-aligned user simulation.** We propose RUSE (Realistic User Simulation Engine) instantiates user simulators along four human behavior dimensions combined into three personas.
- **State-based evaluation.** We evaluate agents through two complementary rule-based verifier sets: *concrete-state* verifiers that inspect whether the final database satisfies all task requirements, and *abstract-state* verifiers that check communication factuality and correct task termination.

We instantiate CRAB-Bench on a trip booking domain with 19 agent tools and 7 user tools, yielding 200 tasks stratified by difficulty (S1–S4). In the hardest stratum (S1), the distractor ratio based on the hard misleading distractors is only 0.05%, and the full search space is more than 500^4.

We evaluate four LLM agents (Claude Sonnet 4.6, DeepSeek V3.2, GLM-5, Qwen3 Coder Next) and find: (i) increasing the number of distractors and task dependencies make the agents struggle with the misleading candidates; (ii) all models suffer consistent performance drops of 19–57% when switching from a generic user simulator to RUSE; (iii) RUSE primarily degrades agents' ability to arrive at correct solutions (concrete-state drops of 17–39%) rather than their conversational quality; (iv) Information Disclosure (D2) is the single most damaging behavioral dimension, and Impatient users, despite adversarial tone, provide useful error signals for problem-solving. We also provide a detailed failure analysis and suggest the possible improvement direction.

2 Related Work

Environments for User-centric Agents. Multi-turn agent benchmarks such as ToolSandbox (Lu et al., 2025), URS (Wang et al., 2024a), MINT (Wang et al., 2024b), τ-bench (Yao et al.), and τ^2-bench (Barres et al.) evaluate agents in simulated service settings with tool use and dynamic user interaction. Recent τ−-Banking (Shi et al., 2026) focuses on unstructured knowledge in the financial domain, stress-testing the joint capability of retrieval and tool use in long-horizon, user-facing interactions. AgentChangeBench (Rana et al.) further studies robustness under mid-dialogue goal shifts. However, these benchmarks share two limitations: they rely on cooperative, template-like user simulators, and they evaluate against fixed ground-truth solutions, making them unsuitable for tasks with multiple valid outcomes. CRAB-Bench addresses both by combining constraint graph-based task generation with structured distractors and state-based evaluation that explicitly accommodates open-ended solutions.

User Simulation for Conversational Agents. Early user simulation focuses on simulating user agents with different profiles or preferences, such as TravelPlanner+ (Singh et al., 2024) and PRELUDE (Gao et al., 2024). After that, more work focuses on the ambiguity in user interaction, such as Ambig-SWE (Vijayvargiya et al., 2025) and CharEval (Li et al., 2026). UserBench (Qian et al., 2025) introduces three core factors for realistic user simulators, who should share the information in an underspecified, incremental, and indirect way. However, recent studies also show that there are still significant gaps between the simulated users and the real-world human users, making the evaluation results less realistic. Seshadri et al. find that evaluations using simulated users systematically miscalibrate: underestimating performance on challenging tasks and overestimating it on moderately difficult ones. Zhou et al. measures alignment between human and 31 LLM-based simulated users, finding that general LLM capability does not reliably translate to faithful user simulation. Together, these results motivate the user simulation design choices of CRAB-Bench: multiple user profiles with human-like interactive behaviors.

3 Constraint-based Realistic Agent Benchmark

3.1 Task Formulation

Similar to τ-bench (Yao et al.), the tasks in our benchmark have a three-component structure: a **user simulator** $\mathcal{U}$, an **agent**, and an **environment** backed by a database $\mathcal{B}$ of the booking systems and a user private database of wallet and card $\mathcal{B}_u$. The agent is equipped with a tool set $\mathcal{T}_a$ and the user simulator with a tool set $\mathcal{T}_u$. As shown in Figure 2, the user simulators use $\mathcal{T}_u$ to interact with $\mathcal{B}_u$, and provide necessary information for booking. The agent books trips based on the user's request with $\mathcal{T}_a$ and modifies the booking system database $\mathcal{B}$.

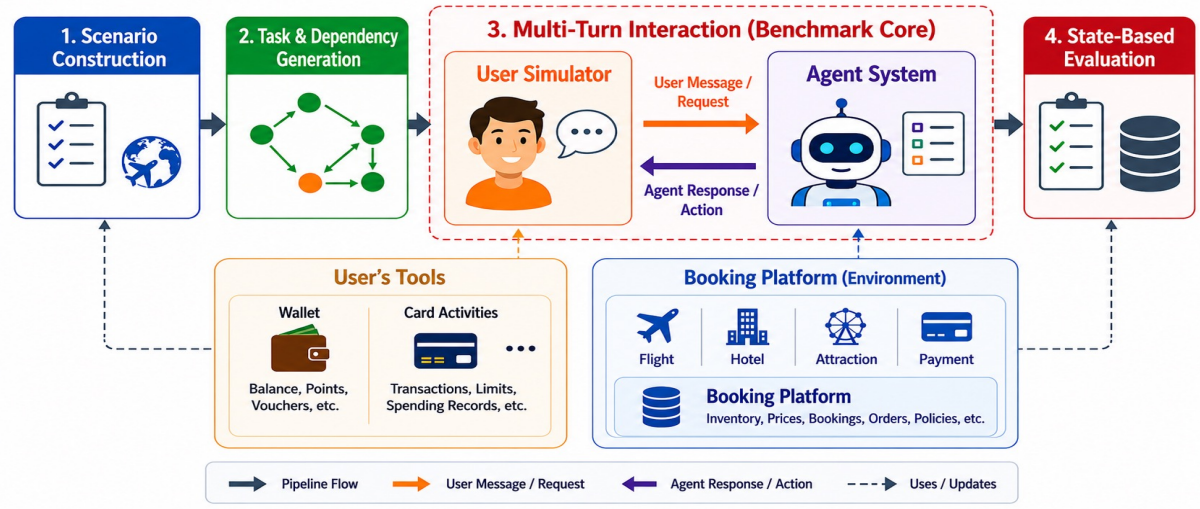

Figure 2: CRAB-Bench Overview. User simulators interact with agent systems with their requests, and the agent uses diverse tools to solve the task. The final solution is verified based on the database state and the communication state.

3.2 Constraint Graph-Based Task Generation

As shown in Figure 1, one trip booking task includes multiple subtasks, such as flight booking and hotel booking. Solving the task requires the agent to satisfy the user requirements in the entity (e.g., flight) in the subtask and resolve the dependencies between entities.

However, it is challenging to initialize the booking system with multiple entities and their dependencies because: (i) there could be multiple valid solutions, making the task easy to solve; (ii) there could be no valid solution at all. Userbench (Qian et al., 2025), which also creates travel planning tasks with multiple domains, solves these by using GPT-4o to generate about 100 combinations as options, and lets the agent choose the correct options. Such a setting reduces the difficulty in solving subtask dependencies, and cannot reflect how realistic trip booking is done.

Instead, CRAB-Bench creates independent databases for each entity and forces the agent to identify dependencies and determine the valid combinations of trips from the huge search space. To solve the above challenges, we propose a constraint-graph framework. It can construct an initial database state $\mathcal{B}_0$ that includes (i) at least one solution, (ii) multiple *distractors*, which are confusing options that only partially satisfy entity or inter-entity constraints to mislead the agent.

3.2.1 Nodes, Objects, and Constraints

Each required entity in the task is represented as a **node**. A node defines a set of discrete-valued

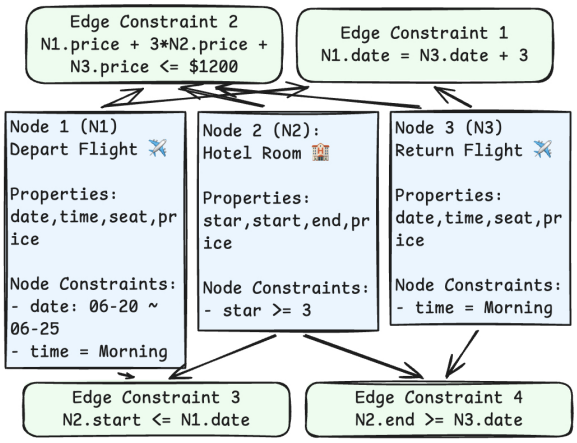

Figure 3: An example constraint graph. The user requirements are: A trip from Chicago to Pittsburgh. Flexible to leave between 06-20 and 06-25. Stay for 3 nights. Morning flights, a hotel $\geq$ 3-stars, budget of $1200.

properties, each with a finite domain of possible values. For example, a FLIGHT node has properties such as `departure_time` $\in$ {MORNING, MIDDAY, NIGHT}, `seat_position` $\in$ {WINDOW, AISLE, MIDDLE}, `seat_type` $\in$ {ECONOMY, BUSINESS}, and etc. A **concrete object** is a complete assignment of values to all properties of a node. A task graph $G = (\mathcal{N}, \mathcal{C})$ collects all nodes $\mathcal{N} = \{n_1, \ldots, n_k\}$ needed for the task and a set of constraints $\mathcal{C}$ that specify which combinations of concrete objects constitute a valid solution.

Constraints in $\mathcal{C}$ fall into two classes. **Domain constraints** are among the properties within a specific domain. For example, a *business-class* seat cannot have a *middle* position. These constraints

are shared across all tasks. **Validity constraints** encode task-specific requirements related to the user's preferences, including node-level and edge-level. **Node-level** validity constraints restrict properties of a single node. For example, in Figure 3, the user asks for a morning flight, so both Node 1 and Node 3 have $time =$ Morning. **Edge-level** validity constraints involve properties across two or more nodes and capture cross-object coherence requirements. In Figure 3, the user's budget is $1200, so the total price of three nodes should not exceed this budget (Edge Constraint 2). A valid solution is a tuple of concrete objects that satisfy all the constraints in $\mathcal{C}$.

3.2.2 Seed Solution Generation

To ensure there is at least one valid solution, we first generate several **seed solutions**, which are valid solutions that satisfy all constraints. When there are multiple seed solutions, cross-combinations of their objects may also form valid solutions, as shown in the dotted lines in Figure 1.

To generate a seed solution, we formulate a CSP over the properties referenced by the edge constraints and sample a solution from it. With these values fixed, each node is solved independently under its local domain and node-level constraints, and one concrete object per node is sampled at random.

3.2.3 Distractor Generation

Based on the validity constraints, we generate two classes of distractors, D_{node} and D_{edge}, that inflate $\mathcal{B}_0$ without introducing additional valid solutions.

Node Distractors (D_{node}). For each node and each node-level validity constraint c, we enumerate all realizable concrete objects that *violate* c while satisfying all domain constraints. For example, for a task where the user requires a morning flight, this yields many flight objects at every other time of day, and over all other combinations of properties which are individually well-formed but directly fail a task requirement.

Edge Distractors (D_{edge}). A valid flight may have no compatible hotel in the database: for instance, no hotel with a matching check-in date. We call such objects edge distractors: individually valid, but unable to form a complete solution with any other object.

We generate edge distractors as follows. For each edge constraint c, we maintain a pool of *valid profiles*, which are the edge-relevant property values present in the seed solutions. A profile is *universally bad* if it is incompatible with every profile currently in the pool from all other nodes. For example, a hotel at $400/night is universally bad when every flight in the pool costs $350 and the user's budget is $700, since no valid pairing exists.

We iteratively generate edge distractor objects by choosing a universally bad profile, fixing its edge property values, and solve a per-node CSP to produce all concrete objects that satisfy the node's domain and node-level constraints. These objects are added to the pool as edge distractors. Then, we recompute which profiles remain universally bad and repeat until none are left. Details are in A.4.

3.2.4 Database Initialization

The output of the pipeline is a flat list of concrete objects tagged as seed solutions, node distractor (D_{node}), or edge distractor (D_{edge}). A **instantiator** converts each object into one or more initialization actions that add the corresponding record into the database. The full sequence of initialization actions executed on the database $\mathcal{B}$ produces the initial database $\mathcal{B}_0$ for the task. For the user database $\mathcal{B}_u$, we randomly initialize it with user and card information and ensure that at least one card has sufficient budget for the task.

3.3 User Simulation Grounded by Humans

We design 4 user behaviors and 3 user personas to narrow the gap between real-world human users and user simulators. Prompts are listed in A.2

User Behavior Dimensions. We follow the same behavior dimensions introduced in Zhou et al. and to design 4 user behaviors for simulation: (D1) `Communication style`: how the user speak to the agents, such as their emotion, tone, the length of sentences, etc. (D2) `Information Disclosure`: how the user shares information; (D3) `Clarification`: how confident the user is when sharing information (D4) `Error Reaction`: how the user reacts to the agents' error.

User Personas. We design 3 user personas that affect communication with agents: `Terse`, `Neutral`, and `Impatient`. `Terse` users communicate in short, clipped sentences, and get straight to the point with minimal information. `Impatient` users are less tolerant of too many questions and mistakes. `Neutral` users are the most collaborative ones that are kind. They always provide detailed

information when asked. Each persona includes different choices of behavior dimensions. Different from τ^2- bench, which simulates users based on their age and background, our simulation focuses on human behavior patterns.

3.4 State-based Evaluation

There may be plenty of valid solutions that satisfy the user requirements in trip booking, making evaluation challenging. Therefore, instead of comparing the solution with the ground-truth set, we design a state-based evaluation, introducing the concrete state (e.g., booking system database B and user private database B_u) and the abstract state (e.g., conversation). Concrete-state verification is a set of task-specific verifiers to test whether user requirements are satisfied in the final database. Abstract-state verification tests two properties: **factuality** (the agent's actions align with its stated plan, e.g. the booked flight ID and price match what was communicated) and **completion** (the interaction ends correctly, e.g. the agent offers a human transfer instead of leaving pending bookings unresolved).

We view a task as solved only if the solution passes all verifiers. If the agent books a valid solution that differs from what it mentioned, it still fails because the solution cannot pass the factuality verifier. If the agent books a valid solution that differs from what it mentioned, it still fails because the solution cannot pass the factuality verifier.

Table 1: `CRAB-Bench` statistics. For each task, we have 4 nodes for 4 required entities and 6 edges for the task dependencies. Distractor ratio is $\frac{\#Valid}{|D_{node}|+|D_{edge}|}$, indicating difficulty in figuring out the solution.

# Seed	# Valid	$\lvert D_{node}\rvert$	$\lvert D_{edge}\rvert$	Distractor Ratio
1	1.0	1819.3	214.8	0.05%
2	7.5	1819.3	204.3	0.37%
3	26.6	1781.2	226.1	1.32%
4	52.9	1780.0	228.9	2.62%

3.5 `CRAB-Bench`

We focus on trip booking because it involves multiple subtasks with dependencies. The 4 required entities in our benchmarks are *departure flight, return flight, hotel, and attraction*. There are 6 types of edge constraints between nodes. The agent tool set $\mathcal{T}_a$ includes 19 tools across 5 categories, and the user tool set $\mathcal{T}_u$ includes 7 tools from 6 categories. The full set is in Table 6.

With our automatic constraint graph-based task generation, given the properties, we can generate full combinations for all their possible values with random seed solutions. To keep our benchmark a reasonable size for evaluation, we create 4 sets with 1 to 4 seed solutions, generate full combinations for properties, and filter these tasks based on the process mentioned in Appendix A.3.

Finally, we keep 50 tasks for each seed solution group, naming them as S1, S2, S3, and S4, resulting in 200 tasks in total. S1 indicates the set of tasks starting from one valid seed solution. The detailed statistics are listed in Table 1. With more seed solutions, the combinations of their different parts lead to more valid solutions. While the number of node distractors and edge distractors is similar, this increases the distractor ratio towards all possible solutions. This may make it easier for the agent to find the correct solutions. Therefore, we assume that S1 is the most difficult subset, while S4 is the easiest with the highest valid rate.

Note that we can easily increase the number of tasks and extend it to other properties and domains based on our algorithm, making it easy to scale up.

4 Experimental Results

In this section, we focus on two research questions: (i) how different agents perform on `CRAB-Bench`? (ii) how `RUSE` affect the agent's performance?

Experimental Settings We follow the $\tau-$ benchmark framework (Yao et al.; Barres et al.) under its MIT License. We use `Claude Sonnet 4.6`, `GLM-5` (744B-A40B MoE), `Qwen3 Code Next` (80B-A3B MoE) and `DeepSeek V3.2`(685B-A37B MoE) as the agent. We use `Claude Sonnet 4.6` for the user simulation. We implement the human-aligned user simulator `RUSE` as described in Section 3.3 and use both the user persona (sampled uniformly) and all behavior dimensions. We also implement the generic user simulator, following τ^2-bench. We use the default inference and reasoning setting and report pass@1 in Table 2 and discuss passˆk in Section A.6.

DeepSeek V3.2 shows the best performance with 0.61, while Qwen3 Coder Next performs worst with 0.20. Although DeepSeek V3.2 provide promising results, its pass@1 on `CRAB-Bench` is much lower that on τ^2- bench (78.9%). Similarly, Claude Sonnect 4.6 shows 79.5% and GLM-

Table 2: Results on CRAB-Bench and the breakdown verification. One solution passes only if it passes both concrete and abstract-state verification. Generic indicates the default user simulator without the user persona and behaviors. Δ indicates the performance drop compared with the generic user simulator.

	Pass Rate			Concrete-state Verification			Abstract-state Verification		
	Generic	RUSE	Δ	Generic	RUSE	Δ	Generic	RUSE	Δ
Claude Sonnet 4 6	0.52	0.32	-38%	0.76	0.62	-19%	0.62	0.60	-3%
GLM-5	0.64	0.44	-31%	0.76	0.56	-27%	0.70	0.65	-6%
Qwen3 Coder Next	0.45	0.20	-57%	0.47	0.28	-39%	0.63	0.63	1%
DeepSeek V3.2	0.75	0.61	-19%	0.88	0.73	-17%	0.79	0.79	0%

5 gains 98.2% on τ^2- bench Telecom domain [1], but none of them get a pass@1 higher than 0.5 on CRAB-Bench with our RUSE. This indicates CRAB-Bench is more difficult, benefiting from the constraints and dependencies in our benchmark.

Introducing human-like RUSE leads to consistent performance degradation across all agents. The pass rate drops from -19% to -57%. This indicates that while agents show a promising performance with generic user simulators, they cannot interact with human-like users reliably. This gap affects agents' capabilities in realistic scenarios. However, for a more powerful model (with a higher pass@1), such as DeepSeek V3.2, the influence is smaller. This indicates that further improving the agent's problem-solving capability may help it better interact with the human users.

RUSE primarily degrades agents' ability to arrive at correct solutions, rather than their conversational quality. Compared with the significant drop in concrete-state verification (17–39%), which focuses on whether the agent can arrive at the correct solution, abstract-state verification is relatively stable (-6% to $+1\%$). This indicates that human-like behaviors make the task more difficult for agents to solve, for example, by changing the way information is shared. However, these user behaviors have less impact on how agents interact with the user, such as how factual the agents are.

5 Analysis

Stronger models can better leverage alternative paths when available. We evaluate concrete-state performance across four task complexity levels, focusing on how the number of valid solutions affects the difficulty of arriving at the right database state. As shown in Figure 4, in general, the pass rate increases from S1 (hard) to S4 (easy). However, compared with DeepSeek V3.2 and Claude Sonnet 4.6, Qwen3 Coder Next does not benefit as much from the increase in valid solutions until the number of valid solutions becomes very large (S4). This indicates that stronger models are better at exploration of alternative paths when available, while weaker models remain bottlenecked.

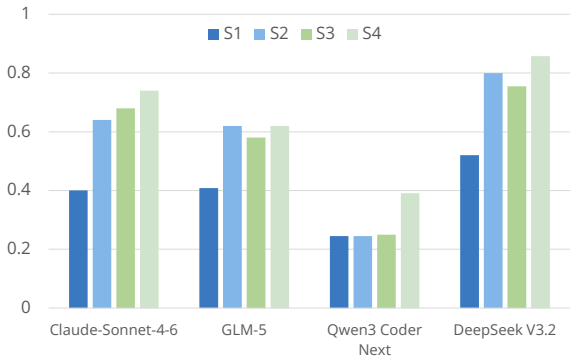

Figure 4: Concrete-state verification pass rate. S4 indicates 4 seed solutions.

More edge constraints lead to substantially harder tasks, independent of distractor pool size. To investigate the effect of different edge constraints, we split CRAB-Bench by date flexibility: whether the user's departure date is a single fixed day or can fall anywhere within a 5-day window. Both groups expose the agent to a comparable distractor pool ($\sim$2,066 objects for fixed-date vs. $\sim$1,999 for flexible-date tasks). As shown in Figure 5, the difference in edge constraints alone produces large drops across all models: pass@1 on flexible-date tasks ranges from 8.3% to 33.3%, compared to 26.9%–61.5% on fixed-date tasks. Even DeepSeek V3.2, the strongest model overall, achieves only 33.3% on the flexible-date subset, isolating multiple subtask dependencies as a primary bottleneck for current agents.

[1] https://benchlm.ai/benchmarks/tau2Bench

Table 3: Communication and efficiency performance on CRAB-Bench. ↑ indicates the higher the better.

	Communication				Efficiency					
	Factuality ↑		Completion ↑		Redundant Calls ↓		Call Cancellation ↓		Admitted Errors ↓	
	Generic	RUSE	Generic	RUSE	Generic	RUSE	Generic	RUSE	Generic	RUSE
Claude Sonnet 4.6	89.6%	86.6%	95.8%	96.3%	3.4%	4.2%	0.1%	0.2%	2.7%	0.3%
GLM-5	91.7%	88.7%	94.5%	92.7%	3.8%	6.9%	0.6%	1.4%	3.1%	1.6%
Qwen3 Coder Next	88.1%	91.1%	93.7%	89.3%	12.8%	15.8%	0.6%	2.1%	8.2%	3.1%
DeepSeek V3.2	97.0%	96.4%	96.0%	95.7%	6.6%	12.6%	0.2%	0.4%	4.6%	1.1%

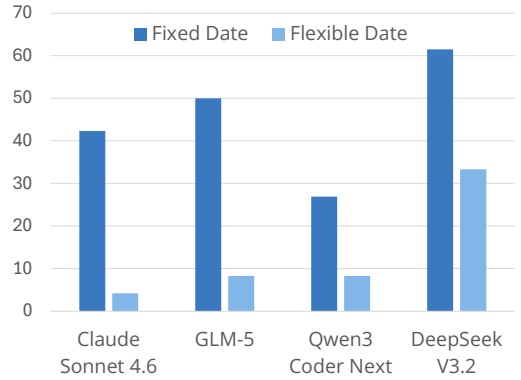

Figure 5: Pass@1 on fixed-date vs. flexible-date tasks. Flexible-date tasks have on average 5.5 active edge constraints vs. 2.5 for fixed-date tasks, with comparable distractor pool sizes.

Agents are less efficient towards RUSE, but they are less likely to admit mistakes. We also investigate efficiency metrics to analyze the agent's performance, including the number of redundant calls (tool call with the same parameters), the number of cancellations during booking (e.g., the agent may book a wrong flight and then cancel), and the number of admitted errors (such as saying '*I made a mistake*'). The detailed definitions are listed in Appendix A.1. In Table 3, factuality and completion are less affected by human-like behaviors, which is consistent with previous findings. However, the efficiency drops significantly among all agents. Redundant tool calls increase uniformly under the human-like condition, especially for `DeepSeek V3.2` and `Qwen3-Coder`. Call cancellation rates roughly double across all models. This indicates that, instead of asking clarification questions before booking, agents usually take action based on the underspecified requirement and then undo prior actions. Interestingly, while there are more call cancellations, admitted errors *decrease* under the human-like condition for all models. This counterintuitive finding suggests that when interacting with more realistic users, agents are less likely to explicitly acknowledge mistakes. They are more likely to mask errors through implicit corrections or simply proceeding without disclosure, a behavior pattern that merits further investigation from a transparency perspective.

Information Disclosure (D2) is the most challenging behavior. We isolate the contribution of each behavioral dimension and persona to the overall difficulty introduced by the human-like user simulator. Starting from the generic user, we independently add persona descriptions and individual behavioral dimensions D1–D4. As shown in Figure 7, assigning a persona alone causes substantial drops across all models. Poor performance in *Information Disclosure* indicates that how users expose information to the agent significantly affects the agent's capabilities. D1 (*Communication Style*) is most damaging to Claude Sonnet ($0.48 \rightarrow 0.20$), while D3 (*Clarification*) and D4 (*Error Reaction*) have comparatively mild effects.

Persona affects agent efficiency more than task success, and `Terse` significantly degrades efficiency. As shown in Figure 6, `Terse`, who provide minimal information, increases the redundant tool calls and admitted errors, compared with `Natural`. The agents are forced to probe the environment when they cannot confirm details through conversation. `GLM-5` doubles its Call Cancellation with `Terse` vs the other two. This indicates that without user elaboration, it books prematurely, then corrects, making booking inefficient. Despite expressing frustration, `Impatient` users don't degrade task success. They actually signal problems clearly (e.g., 'you made an error'), which may help the agent recover.

Failure modes reveal systematic gaps in tool use, constraint reasoning, and transparency. Table 4 and Table 5 summarize common failure modes. Among concrete-state failures, **payment**

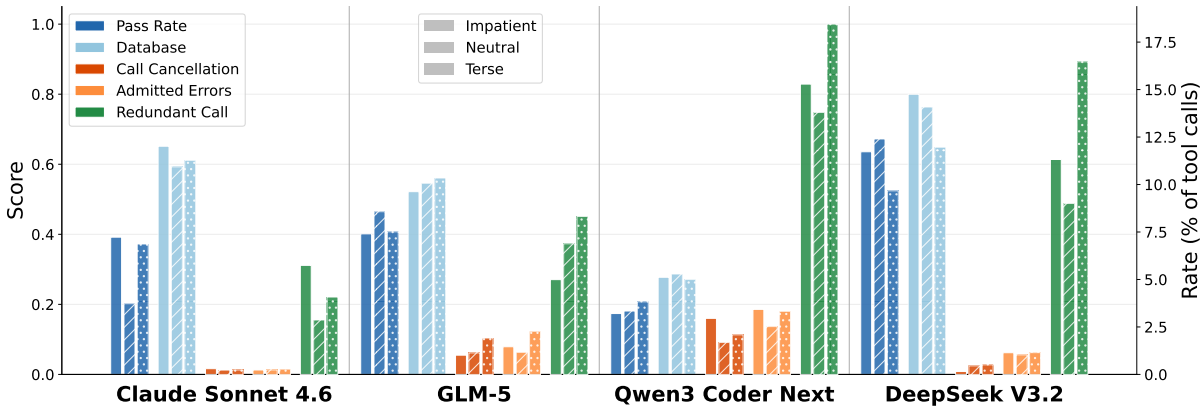

Figure 6: Performance analysis for different personas. The left y-axis is for Pass Rate and Database (concrete-state verification), and the right y-axis is for the rest.

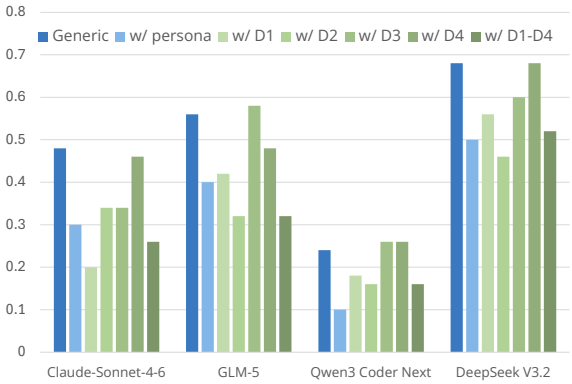

Figure 7: Pass@1 of different types of user simulation. Results are from `Claude Sonnet 4.6` on the S1 subset.

Table 4: Concrete-state failure modes (count=356).

Failure Mode	Key Insight
Payment inability (37%)	Fails to handle payment
Wrong items booked (23%)	Trip violates requirements
Compromised users (25%)	User gives up early
Partial booking (6%)	Trip only partially fulfilled

Table 5: Abstract-state failure modes. Factuality: 280; Completion: 102.

Category	Failure Mode
Factuality	Name not disclosed (56%)
	Summary amount $\neq$ actual charge (16%)
	Booked items $\neq$ approved plan (12%)
	ID mismatch (stated vs. booked) (11%)
	Price mismatch (item-level or pre-charge) (4%)
Completion	No transfer offered for partial booking (96%)

inability dominates, occurring when the agent fails to use payment tools correctly. The second most common failure reflects agents' inability to resolve inter-entity constraints. Among factuality failures, the dominant pattern is agents booking hotels or attractions *without first disclosing the name to the user* (`Claude Sonnet 4.6` accounts for 84 such cases), followed by promise-action mismatches where agents state one plan but execute another (ID mismatch: 11%; plan deviation: 12%). For completion failures, 96% are from agents leaving incorrect bookings uncancelled without notifying the user. Together, these failures point to three directions for improvement: better payment tool grounding, stronger inter-entity constraint propagation, and transparency mechanisms enforcing consistency between agent statements and actions.

6 Conclusion

We presented CRAB-Bench and RUSE, an agentic evaluation framework designed to expose the gap between benchmark performance and real-world agent capability. CRAB-Bench generates tasks with complex inter-entity dependencies and structured distractors, while RUSE replaces idealized user simulators with behavior grounded in human studies. Together, they reveal that frontier LLM agents, despite near-perfect scores on existing benchmarks, struggle substantially when faced with realistic constraints and imperfect users. Our analysis shows that the performance gap is not about conversation quality, which remains largely intact, but about the agent's ability to reason over constrained solution spaces under ambiguous and incremental information disclosure. CRAB-Bench and RUSE can serve as a foundation for developing agents that are robust not only to hard tasks, but to the full complexity of real human interaction.

Limitations

CRAB-Bench is currently instantiated in the trip booking domain, which, while rich in subtask dependencies, may not capture failure modes specific to other agentic settings such as coding assistance. Extending the constraint graph framework to new domains only requires defining entity schemas and constraint types, making it easy to instantiate in new domains. RUSE covers four behavioral dimensions grounded in a prior human study, but real user behavior is broader and more variable. Dimensions such as deception, topic drift, or highly domain-specific jargon are not modeled. In addition, although RUSE is grounded in behavioral observations from human studies, the simulators are still LLM-based and inherit the calibration limitations documented in prior work (Seshadri et al.; Zhou et al.).

References

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

Wei-Lin Chiang, Lianmin Zheng, Ying Sheng, Anastasios Nikolas Angelopoulos, Tianle Li, Dacheng Li, Banghua Zhu, Hao Zhang, Michael Jordan, Joseph E Gonzalez, and 1 others. Chatbot arena: An open platform for evaluating llms by human preference. In *Forty-first International Conference on Machine Learning*.

Ge Gao, Alexey Taymanov, Eduardo Salinas, Paul Mineiro, and Dipendra Misra. 2024. Aligning llm agents by learning latent preference from user edits. *Advances in neural information processing systems*, 37:136873–136896.

Zhicheng Guo, Sijie Cheng, Hao Wang, Shihao Liang, Yujia Qin, Peng Li, Zhiyuan Liu, Maosong Sun, and Yang Liu. 2024. Stabletoolbench: Towards stable large-scale benchmarking on tool learning of large language models. *Preprint*, arXiv:2403.07714.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*.

Philippe Laban, Hiroaki Hayashi, Yingbo Zhou, and Jennifer Neville. 2026. Llms get lost in multi-turn conversation. In *The Fourteenth International Conference on Learning Representations*.

Jialin Li, Yuan Wu, and Yi Chang. 2026. Clareval: A benchmark for evaluating clarification skills of code agents under ambiguous instructions. *arXiv preprint arXiv:2603.00187*.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, and 1 others. Agentbench: Evaluating llms as agents. In *The Twelfth International Conference on Learning Representations*.

Yaxi Lu, Shenzhi Yang, Cheng Qian, Guirong Chen, Qinyu Luo, Yesai Wu, Huadong Wang, Xin Cong, Zhong Zhang, Yankai Lin, and 1 others. 2025. Proactive agent: Shifting llm agents from reactive responses to active assistance. In *International Conference on Learning Representations*, volume 2025, pages 47431–47457.

Cheng Qian, Zuxin Liu, Akshara Prabhakar, Zhiwei Liu, Jianguo Zhang, Haolin Chen, Heng Ji, Weiran Yao, Shelby Heinecke, Silvio Savarese, and 1 others. 2025. Userbench: An interactive gym environment for user-centric agents. *arXiv preprint arXiv:2507.22034*.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, and 1 others. Toolllm: Facilitating large language models to master 16000+ real-world apis. In *The Twelfth International Conference on Learning Representations*.

Manik Rana, Calissa Man, Anotida Expected Msiiwa, Jeffrey Paine, Kevin Zhu, Sunishchal Dev, Vasu Sharma, and 1 others. Agentchangebench: A multi-dimensional evaluation framework for goal-shift robustness in conversational ai.

Preethi Seshadri, Samuel Cahyawijaya, Ayomide Odumakinde, Sameer Singh, and Seraphina Goldfarb-Tarrant. Lost in simulation: Llm-simulated users are unreliable proxies for human users in agentic evaluations.

Quan Shi, Alexandra Zytek, Pedram Razavi, Karthik Narasimhan, and Victor Barres. 2026. τ-knowledge: Evaluating conversational agents over unstructured knowledge. *arXiv preprint arXiv:2603.04370*.

Harmanpreet Singh, Nikhil Verma, Yixiao Wang, Manasa Bharadwaj, Homa Fashandi, Kevin Ferreira, and Chul Lee. 2024. Personal large language model agents: A case study on tailored travel planning. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pages 486–514, Miami, Florida, US. Association for Computational Linguistics.

Mark Vero, Niels Mündler, Victor Chibotaru, Veselin Raychev, Maximilian Baader, Nikola Jovanović, Jingxuan He, and Martin Vechev. 2025. Baxbench: Can llms generate correct and secure backends? *arXiv preprint arXiv:2502.11844*.

Sanidhya Vijayvargiya, Xuhui Zhou, Akhila Yerukola, Maarten Sap, and Graham Neubig. 2025. Interactive agents to overcome ambiguity in software engineering. *arXiv preprint arXiv:2502.13069*.

Jiayin Wang, Fengran Mo, Weizhi Ma, Peijie Sun, Min Zhang, and Jian-Yun Nie. 2024a. A user-centric multi-intent benchmark for evaluating large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 3588–3612.

Xingyao Wang, Zihan Wang, Jiateng Liu, Yangyi Chen, Lifan Yuan, Hao Peng, and Heng Ji. 2024b. Mint: Evaluating llms in multi-turn interaction with tools and language feedback. In *International Conference on Learning Representations*, volume 2024, pages 32593–32627.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik R Narasimhan. {τ}-bench: A benchmark for \underline{T}ool-\underline{A}gent-\underline{U}ser interaction in real-world domains. In *The Thirteenth International Conference on Learning Representations*.

Chen Zhang, Xinyi Dai, Yaxiong Wu, Qu Yang, Yasheng Wang, Ruiming Tang, and Yong Liu. 2025. A survey on multi-turn interaction capabilities of large language models. *arXiv preprint arXiv:2501.09959*.

Xuhui Zhou, Weiwei Sun, Qianou Ma, Yiqing Xie, Jiarui Liu, Weihua Du, Sean Welleck, Yiming Yang, Graham Neubig, Sherry Tongshuang Wu, and 1 others. Mind the sim2real gap in user simulation for agentic tasks.

A Appendix

A.1 Verifiers and Metrics

We use concrete-state verification to check the final state of the database, and use abstract-state verification to check the communication quality. One solution needs to pass both verifications to be correct.

Abstract-state verification includes *Factuality* and *Completion*. Factuality verifies whether the agent's actions align with what they say, and Completion verifies whether the agent terminates the conversation in the right way. Both types of verifiers are rule-based and only have the outcome of True or False.

Factuality verifiers. We inspect the factuality from the following aspects. Note that the verifier is only triggered when the conversation mentions the specific item. For example, if the agent does not provide a post-booking summary, the corresponding verifier is omitted.

- **Pre-charge totals**: the dollar amounts stated before making a charge must match the actual charges
- **Post-booking summary**: the final total must match the confirmed bookings of the database
- **Booking IDs**: flight/hotel/attraction IDs passed to book_* tools must have been mentioned to the user beforehand
- **Booking names**: hotel/attraction names must have been communicated before booking
- **Item-level prices**: per-item prices stated by the agent must match actual charges
- **Approved plan match**: items actually booked must match the IDs the user explicitly approves

Completion verifiers. Following the τ^2-bench, we have three final conversation states: STOP, TRANSFER, OUT-OF-SCOPE. OUT-OF-SCOPE is reserved because we ensure that all tasks are solvable with the given tools and initial database. Therefore, Completion verifiers include:

- Never end with OUT-OF-SCOPE
- If the trip is fully booked with all required entities (flights, hotel, attractions), it should end with STOP (successfully complete) or TRANSFER (fail and need help from human agents).
- If the trip is partially booked, the agent must offer a transfer to a human agent
- If no booking at all, the conversation should end with STOP (e.g., the user gives up).

Efficiency Metrics Furthermore, we investigate the efficiency of communication from tool calls and agent mistakes. Note that these metrics are only for analysis purposes and will not affect the pass rate.

- Total tool calls: all tool calls made by the agent
- Call Cancellation: Unnecessary booking changes, such as book an incorrect flight and then canceling it.
- Admitted errors: messages mention "I apologize for the error", "my mistake", etc.
- Failed Tool Calls: tool call fails
- Redundant Calls: exact-duplicate calls with the same name and arguments.

A.2 User Simulation

User Persona Prompt

```
Here are the 3 personas and 4 behavior
dimensions we used in CRAB-Bench.
Terse
You are a no-nonsense person who communicates
in short, clipped sentences. You rarely
use pleasantries and prefer to get straight
to the point. You answer questions with
the minimum information needed and do not
elaborate unless pressed.

Neutral
You are a straightforward person. You
are polite but not overly friendly. You
provide information when asked without much
embellishment. Your tone is conversational
but efficient.

Impatient
You are in a hurry and slightly irritable. You
want the booking done quickly. If the agent
asks too many questions or makes errors, you
become noticeably curt. You do not apologize
for the agent's mistakes and you express
brief frustration when things go wrong.
```

Behavior Demension Prompt

```
(D1) Communication style: keep your replies
short - fragments and single sentences are
fine. Do not pad messages with pleasantries
like 'Sure!', 'Thank you so much!', or 'That
sounds great!'. Vary your tone naturally;
you may be curt, neutral, or brief. Never
open with a greeting longer than one short
sentence.

(D2) Information disclosure: only share
information from the 'Say upfront' section in
your first message. Reveal other details only
when the agent asks for them or when they
become relevant. Do not volunteer all your
requirements at once.

(D3) Clarification: state your preferences
plainly when asked. Do not stack multiple
hedges or dramatic uncertainty markers
(e.g., 'I'm crossing my fingers', 'maybe I'm
cursed'). At most one brief hedge per turn
is acceptable (e.g., 'I'm not sure about the
dates').

(D4) Error reactions: if the agent makes
a mistake or retrieves wrong information,
correct it tersely (e.g., 'No, wrong dates.'
or 'That's not it.'). Do not apologize for
the agent's error, do not cooperatively
pivot to a different strategy, and do not
volunteer alternative information to help the
agent recover. If the agent repeats the same
mistake, you may express brief frustration.
```

A.3 Benchmark details

Information Control for User Simulation To control how the user simulator provides information to the agent, we split the user's known information into two parts: one set of information is the basic information, which can be said upfront; the other set of information is detailed information, which should only be provided when asked. For example, as shown in Figure 1, when booking a trip, the departure day and the duration can be provided in the first round. The other information, such as the name, the traveler's date of birth, and the traveling preferences, will not be revealed until the agent explicitly asks. We use this two-level information sharing as the default behavior in the user personas setting. In the behavior dimensions setting, we implement this for `Information Disclosure`.

Diversity and Complexity Balance in `CRAB-Bench` After creation, we first de-duplicate the seed solutions and then sort the tasks based on the number of valid solutions, D_{node} and D_{edge}. Typically, the task with fewer valid solutions and more distractors requires the agent to search more, making it more difficult. Finally, we use a round-robin across strata to balance the difficulty and diversity. We pick the next-hardest task in each group in turn. The groups are split based on important properties that will affect booking, such as the number of passengers, the budget, the date flexibility, the seat preference, and hotel star ratings. the distribution of properties is shown in Figure 8.

Tool Definition We list all tools we use in Table 6. For trip booking, we also implement the modification and cancellation tools in order to recover from incorrect bookings.

Table 6: Tool usage in the Travel domain simulation. R = Read, W = Write, G = Generic.

Tool Name	Owner	Type	Category
get_customer_information	Agent	R	Platform
update_customer	Agent	W	Platform
transfer_to_human_agents	Agent	G	Platform
list_all_airports	Agent	R	Flights
search_flights_by_route	Agent	R	Flights
get_flight_booking_details	Agent	R	Flights
get_price_airline_booking	Agent	R	Flights
search_available_seats	Agent	R	Flights
book_flight_with_seats	Agent	W	Flights
cancel_flight	Agent	W	Flights
search_hotels_by_city	Agent	R	Hotels
search_available_rooms	Agent	R	Hotels
get_price_hotel_booking	Agent	R	Hotels
book_hotel_with_rooms	Agent	W	Hotels
search_attractions_by_city	Agent	R	Attractions
book_attraction	Agent	W	Attractions
get_recent_payment_transactions	Agent	R	Payments
get_transaction_details	Agent	R	Payments
charge_booking	Agent	W	Payments
get_my_payment_cards	User	R	Wallet
set_default_payment_card	User	W	Wallet
get_my_trip_confirmations	User	R	Confirmations
get_trip_spending_summary	User	R	Spending
get_recent_card_activity	User	R	Card Activity
record_payment_approval	User	W	Approvals
add_payment_method_to_platform	User	W	Platform

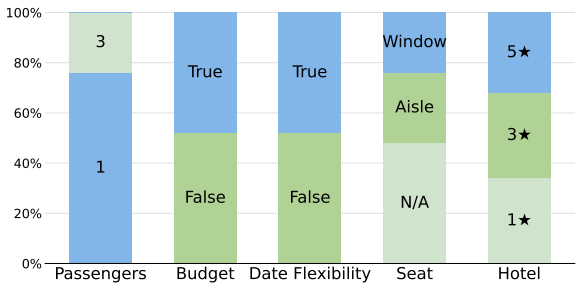

Figure 8: Distribution of the important properties to ensure the diversity of our benchmark.

A.4 Graph Algorithm Details

Notation. Let $G = (\mathcal{N}, \mathcal{C})$ be a task constraint graph. Each **node** $n \in \mathcal{N}$ represents a required entity type (e.g. a flight seat or a hotel room) and is defined by a set of discrete-valued properties, each with a finite domain $D(p)$. The constraint set $\mathcal{C}$ is split into two disjoint subsets:

- $\mathcal{C}_{\text{node}}$: **node constraints**, predicates over properties of a *single* node encoding per-entity task requirements (e.g. departure time must be morning).
- $\mathcal{C}_{\text{edge}}$: **edge constraints**, predicates over properties drawn from *two or more* nodes encoding cross-entity coherence requirements (e.g. $\texttt{flight.price} + \texttt{hotel.price} \leq B$).

An **assignment** for node n is a mapping from each of n's properties to a value in its domain, satisfying n's domain constraints. A **valid solution** is a tuple of assignments, one per node, that jointly satisfy every constraint in $\mathcal{C}$. The **profile** of an assignment o with respect to an edge constraint c, written $\text{profile}(o, c)$, is the sub-tuple of values for the properties of o's node that are referenced by c.

The overall goal of the generation pipeline is to produce a large list of assignments (both seed assignments from valid solutions, and distractors) that are subsequently *materialized* into the initial database $\mathcal{B}_0$ before dialogue begins (see Section A.5).

Helper functions. The algorithms below rely on three abstract helpers:

- $\text{SampleCSP}(\mathcal{V}, \mathcal{C}', k)$: return up to k randomly sampled solutions from the CSP defined over variable set $\mathcal{V}$ with constraints $\mathcal{C}'$.[2]
- $\text{SolveNode}(n, \mathcal{C}', ov)$: return all assignments for node n satisfying its domain constraints, the node constraints $\mathcal{C}'$, and any property values pinned by

[2]Implemented via `python-constraint` (backtracking solver). Two strategies are supported: enumerate all solutions and draw k at random, or shuffle variable domains before each call and extract the first solution, avoiding full enumeration when the solution space is large.

overrides ov.

- SolveGraph(G, k): return up to k solutions from the full graph CSP (used only as a fallback in Algorithm 1).

Subroutine UNIVERSALLYBAD(π, n, c): returns `true` iff profile π for node n violates c for *every* combination of currently valid profiles from all other nodes involved in c:

$$\forall\, n' \neq n,\ \pi_{n'} \in valid[n'] : \\ \text{assemble}(\pi,\ \{\pi_{n'}\},\ c) \in \mathcal{I}_c$$

A.5 Materialization

Once the generation pipeline produces its full list of assignments (seed assignments and distractors), each must be converted into one or more concrete database insertion calls before dialogue begins. This conversion is handled by a **materializer**, which is type-driven: the node type of each assignment selects the appropriate insertion routine, and the property values are translated into the arguments of the corresponding tool call.

For the travel domain, there are three node types.

Flight seat assignments (`flight_seat_set`). Each assignment specifies a route (origin, destination, duration), a departure date, a time of day, a seat type (economy, premium economy, or business), a seat position (window, aisle, or middle), and a price level. Before insertion, all assignments sharing the same (route, departure date, time of day) triplet are grouped into a single physical flight, since they represent different seat configurations on the same service. The materializer issues one `add_flight_with_seats` call per group, passing the route and date once and bundling all seat specifications (type, position, count, and mapped dollar price) as a single list argument.

Hotel room assignments (`hotel_room_availability`). Each assignment specifies a city, a star rating, a room type, a bed type, a maximum occupancy, an availability start date, a number of available nights (`num_dates`), and a price per night. Assignments sharing the same (city, star rating) pair are grouped into one hotel entity. The materializer issues one `add_hotel` call per group, then one `add_room_to_hotel` call per assignment, setting the availability window to [`start_date`, `start_date` + `num_dates`].

Attraction assignments (`attraction_slot`). Each assignment specifies a city, a category (museum, tour, or show), a date, a time of day (morning, afternoon, evening, or all-day), and a price per ticket. Each assignment maps one-to-one to an `add_attraction` call with no grouping. Time-of-day values are translated to a concrete start/end window (e.g. MORNING → 09:00–12:00); all-day events receive a 00:00–23:59 window.

Algorithm 1 GenerateSeedSolutions(G, k, m)

Require: Graph $G{=}(\mathcal{N}, \mathcal{C})$, desired seed count k, attempt multiplier m
Ensure: Set of seed solutions $\mathcal{S}$

```
S ← ∅
V ← properties referenced by any c ∈ C_edge
C_mini ← {c ∈ C | all variables of c are in V}
B ← SampleCSP(V, C_mini, m·k)
for each binding β ∈ B do
    sol ← {}
    ok ← true
    for each node n ∈ N do
        ov ← values in β for properties of n
        O ← SolveNode(n, C_node(n), ov)
        if O = ∅ then
            ok ← false; break
        end if
        sol[n] ← UniformSample(O)
    end for
    if ok and sol ∉ S then
        S ← S ∪ {sol}
    end if
    if |S| = k then break
    end if
end for
if |S| < k then
    S ← SolveGraph(G, k)
        // fallback: full graph CSP
end if
return S
```

A.6 More Experiment Results

Huge gap between pass@k and passˆk indicates models still struggle with solving tasks reliably. With 4 independent trials per task, we report pass@k (at least one success in k trials) and passˆk (all k trials succeed) to measure both capability coverage and behavioral consistency. `DeepSeek V3.2` achieves the highest pass@4, demonstrating that the majority of tasks are within its capability frontier, yet its passˆ4 of 0.12 reveals that only a small fraction of tasks are reliably solved. This gap between coverage and consistency is shared across

Algorithm 2 GenerateNodeDistractors(G, ρ)

Require: Graph $G{=}(\mathcal{N}, \mathcal{C})$, sampling ratio $\rho \in (0, 1]$
Ensure: Node distractor set D_{node}

$D_{\text{node}} \leftarrow \emptyset$
for each node $n \in \mathcal{N}$ **do**
  for each $c \in \mathcal{C}_{\text{node}}(n)$ **do**
    $\mathcal{C}^{-c} \leftarrow \big(\mathcal{C}_{\text{node}}(n) \setminus \{c\}\big) \cup \{\neg c\}$
    // all node constraints, with c negated
    $V \leftarrow \text{SolveNode}(n,\ \mathcal{C}^{-c},\ \{\})$
    // well-formed objects that violate c
    $D_{\text{node}} \leftarrow D_{\text{node}} \cup \text{UniformSample}(V,\ \lceil \rho \cdot |V| \rceil)$
  end for
end for
return D_{node}

Algorithm 3 GenerateEdgeDistractors(G, $\mathcal{S}$)

Require: Graph $G{=}(\mathcal{N}, \mathcal{C})$, seed solutions $\mathcal{S}$
Ensure: Edge distractor set D_{edge}

$D_{\text{edge}} \leftarrow \emptyset$
for each $c \in \mathcal{C}_{\text{edge}}$ **do**
  $\mathcal{I}_c \leftarrow \big\{\mathbf{t} \in \prod_{(n,p)\in c} D(p) \mid c(\mathbf{t}) = 0\big\}$
  // all value tuples violating c
  for each node n in c **do**
    $valid[n] \leftarrow \{\text{profile}(o, c) \mid o \in \text{seed objects of } n\}$
    // init from seed objects
    $P[n] \leftarrow \text{SolveNode}(n,\ \mathcal{C}_{\text{node}}(n),\ \{\})$
    // all node-valid objects
    $bad[n] \leftarrow \{\, \text{profile}(o, c) \mid o \in P[n],\ \text{UniversallyBad}(\text{profile}(o, c),\ n,\ c)\,\}$
  end for
  while $\exists\, n : bad[n] \neq \emptyset$ **do**
    Pick (n, π) with $\pi \in bad[n]$
    $O \leftarrow \text{SolveNode}(n,\ \mathcal{C}_{\text{node}}(n),\ \pi)$
    // fix edge-relevant props to π
    $D_{\text{edge}} \leftarrow D_{\text{edge}} \cup O$
    $valid[n] \mathrel{+}= \{\pi\}$
    $bad[n] \mathrel{-}= \{\pi\}$
    for each other node n' in c **do**
      $bad[n'] \leftarrow \{\pi' \in bad[n'] \mid \text{UniversallyBad}(\pi',\ n',\ c)\}$
      // π may rescue profiles in $bad[n']$
    end for
  end while
end for
return D_{edge}

all models: `Qwen3 Coder Next` exhibits the lowest performance on both metrics.

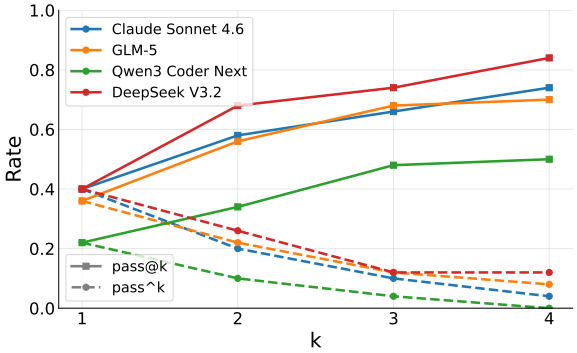

Figure 9: Pass^k and pass@k for different agents.

A.7 Potential Risks

The failure modes identified in our analysis, particularly agents booking items that contradict what they told the user, highlight transparency risks in deployed systems. Benchmarks that do not evaluate factuality between agent speech and action may miss this class of failures entirely, and we encourage future work to treat communication consistency as a first-class evaluation criterion.

A.8 The Use of LLMs

A large language model (LLM) was used as a general-purpose assistive tool to check grammar and correct typographical errors in this paper. The LLM did not contribute to research ideation, experimental design, analysis, or substantive writing. The authors take full responsibility for the content of the paper.