Beyond IVR: Benchmarking Customer Support LLM Agents for Business-Adherence

Sumanth Balaji **Piyush Mishra** **Aashraya Sachdeva**[*] **Suraj Agrawal**
{sumanth.balaji, piyush.mishra, aashraya, suraj.agrawal}@observe.ai
Observe.AI
Bangalore, India

Abstract

Traditional customer support systems, such as Interactive Voice Response (IVR), rely on rigid scripts and lack the flexibility required for handling complex, policy-driven tasks. While large language model (LLM) agents offer a promising alternative, evaluating their ability to act in accordance with business rules and real-world support workflows remains an open challenge. Existing benchmarks primarily focus on tool usage or task completion, overlooking an agent's capacity to adhere to multi-step policies, navigate task dependencies, and remain robust to unpredictable user or environment behavior. In this work, we introduce JourneyBench, a benchmark designed to assess policy-aware agents in customer support. JourneyBench leverages graph representations to generate diverse, realistic support scenarios and proposes the User Journey Coverage Score, a novel metric to measure policy adherence. We evaluate multiple state-of-the-art LLMs using two agent designs: a Static-Prompt Agent (SPA) and a Dynamic-Prompt Agent (DPA) that explicitly models policy control. Across 703 conversations in three domains, we show that DPA significantly boosts policy adherence, even allowing smaller models like GPT-4o-mini to outperform more capable ones like GPT-4o. Our findings demonstrate the importance of structured orchestration and establish JourneyBench as a critical resource to advance AI-driven customer support beyond IVR-era limitations.

Keywords: customer support, large language models, LLM agents, policy adherence, benchmarking, JourneyBench, user journey coverage score

1 Introduction

Customer support automation has traditionally relied on Interactive Voice Response (IVR) systems: automated telephone platforms that gather information and route calls through voice prompts and keypad inputs. While IVR enforces rigid flows via static decision trees to ensure compliance, it often lacks flexibility, resulting in poor user experience and high frustration (Dean, 2008; Coman, 2025). Advances in large language models (LLMs) enable LLM agents: autonomous systems combining textual reasoning and tool-use to handle multi-turn conversations and dynamically manage customer support workflows (Yao et al., 2023; Schick et al., 2023; Wen et al., 2025). Throughout this paper, we use "agent" to refer specifically to these LLM-powered autonomous systems. JourneyBench evaluates agents in text-based conversations, as extension to voice deployments is straightforward with speech-to-text and text-to-speech modules.

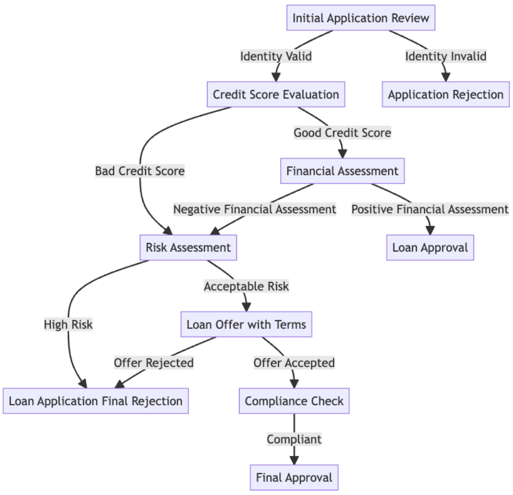


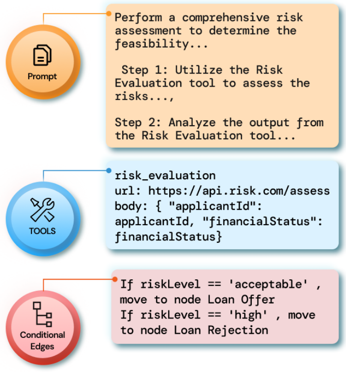
Figure 1: Example SOP graph for loan application processing, showing sequential tasks and decision points.

Ensuring that agents follow business policies and procedural requirements remains a core challenge in production deployments. Standard Operating Procedures (SOPs) are structured workflows that prescribe execution order, validation checks, and exception handling protocols, encoding operational logic and compliance rules. As illustrated in Figure 1, a compliant agent completes all required steps: identity verification, credit evaluation, finan-

---

[*] Corresponding author

Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics
Volume 5: Industry Track, pages 193–208
March 25-27, 2026 ©2026 Association for Computational Linguistics



```
Perform a comprehensive risk
assessment to determine the
feasibility...

 Step 1: Utilize the Risk
Evaluation tool to assess the
risks...,

Step 2: Analyze the output from
the Risk Evaluation tool...
```

```
risk_evaluation
url: https://api.risk.com/assess
body: { "applicantId":
applicantId, "financialStatus":
financialStatus}
```

```
If riskLevel == 'acceptable' ,
move to node Loan Offer
If riskLevel == 'high' , move
to node Loan Rejection
```

Figure 2: Components within a single node: the task description (prompt), available tools for execution, and conditional pathways (edges) that define transitions to the next node based on outcomes.

cial assessment, risk assessment, and loan decision with proper validations, whereas a non-compliant agent may skip risk assessment and proceed directly to approval, achieving the user's goal while violating business logic and creating regulatory and financial risk. Existing benchmarks evaluate goal completion rather than pathway adherence, leaving this gap unaddressed. We therefore use the term **policy-aware agent** to denote an agent that consistently follows prescribed policies throughout the interaction.

We distinguish **tools** (callable functions/APIs for atomic operations, e.g., `GET /customer/{id}`) from **tasks** (higher-level units combining multiple tools, e.g., "identity verification"). Recent benchmarks (Yao et al., 2024; Lu et al., 2025; Trivedi et al., 2024) evaluate tool selection and state transitions, but inadequately assess complete task sequences with complex inter-task dependencies.

To address this gap, we introduce **JourneyBench**, a benchmark for evaluating **policy-aware agents** in customer support. JourneyBench represents SOPs as graphs to generate diverse scenarios, including challenges such as branching logic, missing inputs, and occasional tool failures. It also includes the User Journey Coverage Score (UJCS), which measures how well an agent follows the required sequence of actions defined by an SOP.

Our contributions are:

- A benchmark, **JourneyBench**, for assessing policy-aware agents in customer support using graph-structured SOPs that capture task dependencies and policy constraints.
- The **User Journey Coverage Score (UJCS)**, a metric for measuring adherence to SOP-mandated action sequences.

Benchmark	**Avg Turn**	**Avg Tool Calls**	**Dataset Size**	**Tools**
JourneyBench (ours)	10.91	3.34	703	41
E-commerce	13.37	3.06	232	12
Loan Application	6.57	3.69	230	15
Telecommunications	12.79	3.28	241	14
TOOLSANDBOX (Lu et al., 2025)	13.9	3.80	1032	34
BFCLV3 (Yan et al., 2024)	2.00	0.78	2000	1193
Tau Bench (Yao et al., 2024)	29.33	4.48	165	24

Table 1: Comparison of JourneyBench with other agent benchmarks. JourneyBench statistics are presented overall and broken down by domain.

- An empirical comparison showing that a **Dynamic-Prompt Agent** guided by workflow structure performs more reliably than a **Static-Prompt Agent**, highlighting the value of structured control in business settings.

2 JourneyBench Framework

The JourneyBench framework evaluates policy-aware agents using structured workflow representations. We note that these components can be manually defined or synthetically generated. It consists of four core components: (1) SOP Graphs: Directed Acyclic Graphs encoding business workflows as tasks with conditional transitions; (2) Nodes: individual tasks with natural language descriptions, available tools, and procedural rules for state transitions; (3) User Journeys: specific paths through SOP graphs representing realistic agent-user interactions; and (4) Scenarios: test cases derived from user journeys that assess agent robustness under varied conditions, such as missing inputs or tool failures. Figure 3 illustrates the multi-phase generation process.

2.1 SOP Representation

We model each SOP as a Directed Acyclic Graph (DAG), where nodes represent tasks and edges define valid transitions according to business logic. The DAG encodes task order, decision points, and policy constraints, serving as a blueprint for agent behavior. Figure 1 shows an example SOP graph for loan processing. Henceforth, we use "node" to refer to a task within the graph.

Node Structure: Each node represents a task, including its natural-language description, available tools (e.g., APIs), input/output parameters, and conditional pathways for transitions. Conditional pathways encode procedural rules as logical expressions over tool outputs (e.g., `riskLevel == 'acceptable'`), allowing complex workflow logic to be expressed clearly. During agent execution, these conditions deterministically select the next node, ensuring strict adherence to the SOP. Figure 2 illustrates a node's structure; technical details

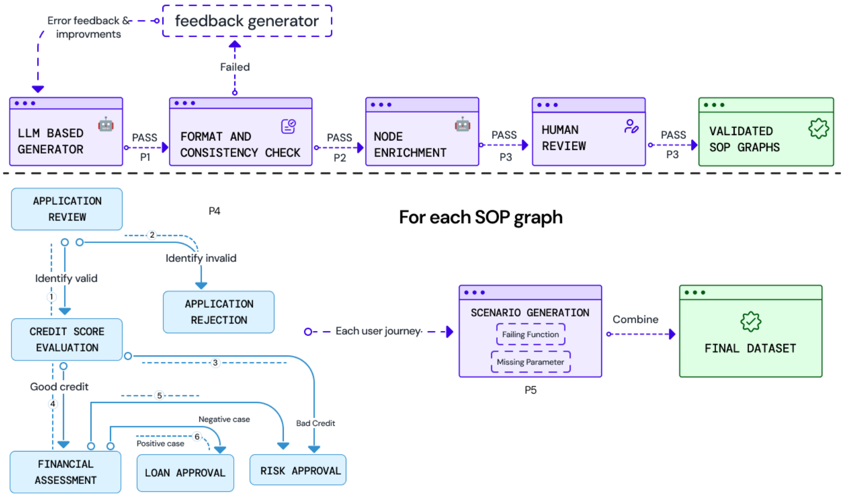


Figure 3: Overview of our data generation pipeline across four phases (P1–P4). Validated SOP Graphs generate numbered user journeys via Breadth-First Search (BFS) traversal, which are then used to create diverse evaluation scenarios for the final benchmark dataset.

are in Appendix A.3.

2.2 Synthetic Dataset Curation

While SOP graphs can be manually authored for specific business processes, constructing a large and diverse benchmark requires a scalable generation pipeline. To build JourneyBench, we automatically generate a dataset of SOP graphs and corresponding interaction scenarios. To minimize human effort, we employ a multi-phase generation process inspired by recent work on LLM-based dataset creation(Barres et al., 2025).

Phase 1: Graph Generation and Refinement A state-of-the-art LLM generates foundational SOP graphs for 10 candidate domains, ensuring workflow complexity and realism. Outputs are validated for acyclicity and connectivity; if issues arise, an iterative LLM-based refinement resolves them (Appendix C). Once validated, node descriptions are enriched with detailed task/tool specifications and examples to reduce ambiguity. This method allows for creative, domain-specific workflows with reduced human effort (Appendix B).

Phase 2: Manual Review Human review ensures logical consistency in workflows, task and tool suitability, and overall graph quality. Each SOP graph is independently reviewed by five contact center agents (domain experts) against three binary pass/fail checks: **Logical Structure** (flow is logically correct and executable end-to-end), **Coherence** (node/tool descriptions and parameters are contextually appropriate and consistent), and **Complexity** (appropriate difficulty for the domain, neither trivial nor needlessly convoluted). A graph is accepted only if all five annotators unanimously pass all three checks ("5-of-5 agreement"). Of 10 candidate graphs, 4 met this standard; three diverse graphs one per domain (Telecommunications, E-commerce, Loan Application) were selected for benchmark experiments. This generate and filter approach enables rapid iteration: generating 10 diverse candidate graphs via LLM took under 1 hour, while manual authoring of comparable graphs would require weeks of expert time. Human review thus serves as a scalability multiplier rather than a bottleneck. See Appendix K for details.

Phase 3: User Journey Generation A user journey is a specific execution path through an SOP graph, representing the sequence of nodes and tool calls a user might follow to achieve their goal. We enumerate all possible paths using Breadth-First Search (BFS) (Figure 3).

Agents are evaluated via simulated conversations, with GPT-4o acting as the user, following established evaluation practices (Yao et al., 2024; Lu et al., 2025). Each simulation uses a **user seed**, a structured prompt specifying: (1) the target journey, (2) user information parameters (e.g., applicant ID), and (3) instructions for natural conversation through the tasks. Example seeds and templates are provided in Appendices A.1 and F.

JourneyBench evaluates workflow adherence rather than tool implementation, treating tools as black boxes with pre-generated responses. For each journey, tool responses that influence workflow branching (e.g., `riskLevel`) are set algorithmically to follow the target path (Appendix D), while other outputs, such as timestamps or confirmation IDs, are generated by an LLM for realism. During evaluation, agents receive these pre-generated responses, ensuring deterministic and reproducible testing. All user journeys are manually reviewed for logical consistency before scenario generation.

Phase 4: Scenario Data Generation From each user journey, we generate multiple evaluation **scenarios**, each representing a full conversational test case with an initial state and expected outcome. The baseline is the "correct context" scenario, where all user parameters are present and tools work as intended. From each correct context case, we systematically construct two additional scenario types: **Missing Parameter**, where required user inputs are withheld and unreachable tool calls are removed from the expected tool trace; and **Failing Function**, where a tool call fails (e.g., API error), and the trace is updated to remove downstream calls that can no longer execute. Duplicate scenarios with identical sequences and responses are removed to ensure uniqueness.

Table 1 shows that JourneyBench offers equal or better coverage across conversational depth, toolset size, and dataset size compared to other benchmarks. It provides a robust benchmark for testing agentic capabilities in policy-driven domains.

3 Evaluation Metrics

To assess an agent's adherence to business workflows, we evaluate its performance on simulated conversational scenarios from user journeys. Each journey specifies a sequence of tool calls and parameters, so our evaluation checks strict procedural adherence and execution accuracy. This forms the basis of our main metric, the User Journey Coverage Score (UJCS).

Tool Trace Alignment: For each simulated conversation, Tool Trace Alignment compares the predicted tool call sequence (T_{act}) with the expected sequence (T_{exp}). Any missing, extra, or misordered tool call indicates an SOP violation, giving the conversation a score of 0.

Tool Call Accuracy: For each simulated conversation, this metric quantifies the correctness of parameter values supplied during tool execution. Tool Call Accuracy TCA_{conv}, score for a single conversation is defined by Equation 1 where $C_i = |P_{act}^{(i)} \cap P_{exp}^{(i)}|$ is the count of correct parameters for the i-th tool call, and $E_i = |P_{exp}^{(i)}|$ is the number of expected parameters, $L = |T_{exp}|$ is the length of the trace, and S_{conv}, score for a single conversation.

$$TCA_{conv} = \begin{cases} \frac{\sum_{i=1}^{L} C_i}{\sum_{i=1}^{L} E_i} & \text{if } T_{act} = T_{exp} \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

User Journey Coverage Score (UJCS): The metric evaluates overall efficacy of an agent for a given SOP graph on N conversations.

$$\text{UJCS} = \frac{1}{N} \sum_{k=1}^{N} TCA_{conv_k} \tag{2}$$

4 Experimentation

4.1 Instantiating Agents with SOP

To study agent adherence to SOPs, we instantiate two variants of agents:

Static-Prompt-Agent (SPA): SPA uses a single, static system prompt. Using a consistent textual template, the entire SOP is transformed into one comprehensive system prompt. The SOP's conditional branching logic is encoded using if-then statements. Tools for all nodes are added to the system prompt as well. An example of this prompt structure can be found in Appendix E.

Dynamic-Prompt-Agent (DPA): The DPA models the SOP as a state machine, processing one node at a time (see Appendix A.3). After each tool execution, an orchestrator: a control component that manages the workflow state and transitions, interprets the response (Appendix A.2), and determines the next node by evaluating conditional pathways defined in the SOP logic. Each transition replaces the previous prompt and updates the accessible tools. This design minimizes context overload, supports mid-flow corrections, and promotes reliable policy execution (see Appendix M).

We exclude explicit planning-based approaches such as ReAct(Yao et al., 2023) due to their significant latency, which makes them unsuitable for real-time interactions in customer support. Additionally, we developed a custom framework for the management of SOP's runtime state and facilitate the handling of conditional pathways. Popular libraries (e.g., LangGraph, CrewAI (LangChain AI, 2023; CrewAI Inc., 2023)) could help construct agents of similar capabilities; we chose a custom framework to ensure reproducibility and stable experimental control across runs and to avoid dependency churn.

	SPA*			DPA†		
Model	**Correct Context**	**Failing Function**	**Missing Parameter**	**Correct Context**	**Failing Function**	**Missing Parameter**
GPT-4o (Wu et al., 2024)	0.871	0.511	0.309	0.873	0.857	0.530
GPT-4o-mini (Wu et al., 2024)	0.720	0.326	0.263	0.718	0.816	0.414
Claude 3.5 Haiku (Anthropic, 2024)	0.234	0.285	0.240	0.504	0.776	0.453
Llama 3.3 (Grattafiori et al., 2024)	0.237	0.264	0.256	0.311	0.345	0.332

Table 2: User Journey Coverage Scores (UJCS) for Dynamic-Prompt-Agent (DPA†) and Static-Prompt-Agent (SPA*) across scenario types. Higher scores indicate better performance.

4.2 Experiments

All experiments use a 40 turn limit and default LLM temperature settings. The simulated user (GPT-4o, Section 2.2) follows the predefined journey while maintaining natural conversation and preventing information leakage. From each correct-context journey, our benchmark generates additional scenarios with failed functions or missing parameters to test agent robustness.

Metrics: Agent performance is evaluated using the **User Journey Coverage Score** (refer to Section 3). We also track the number of successfully completed conversations and various error types.

Real-World Deployment: The structured DPA-based orchestration is deployed in production across client contact centers, reliably handling 6,000+ calls daily while meeting real-time and policy adherence requirements. These production systems process voice calls by converting speech to text, applying the same text-based DPA workflow logic evaluated in JourneyBench, and converting responses back to speech. This operational footprint demonstrates that structured agent control is practical and effective beyond controlled simulations.

Realism Validation (LLM-as-a-Judge): To ensure synthetic conversations reflect production-quality interactions, we evaluate them using the same LLM as a judge rubric applied in client Quality Assurance (QA). The rubric measures *Conversational Proficiency* (CP; e.g., empathy, clarity, turn-taking) and *Goal Attainment* (GA; e.g., intent recognition, request resolution) via binary Yes/No questions aggregated across conversations. Synthetic conversations achieve 84.37% overall (82.33% CP; 87.78% GA), comparable to production QA distributions, indicating that benchmark traces realistically capture agent behavior and policy adherence. Appendix L details rubric based validation.

Results and Analysis: Evaluations on **JourneyBench** demonstrate consistent performance gains with the **Dynamic-Prompt-Agent (DPA)** over the **Static-Prompt-Agent (SPA)**. As shown in Tables 2 and 3, GPT-4o with DPA achieves a UJCS of **0.717**, substantially higher than SPA's 0.564, highlighting the value of explicit workflow guidance for policy adherence. Scenario-based testing further shows that SPA performance drops under disturbances such as failed functions or missing parameters, whereas DPA maintains stable coverage across all scenarios. Notably, GPT-4o-mini with DPA (0.649) outperforms GPT-4o with SPA (0.564), demonstrating that structured orchestration enables smaller, cost-efficient models to match or exceed larger ones.

4.3 Error Analysis

We manually went through conversations where UJCS was low to identify error classes. We group the errors into the following three classes:

Dependency Violations: Dependency violations occur when an agent proceeds without required parameters or prior tool use, violating SOP logic. **SPA** often advanced despite missing inputs or failures, while **DPA** correctly halted to maintain logical consistency. More examples are in Appendix G.

Hallucination in Parameter Values: Parameter hallucination occurs when an agent uses example values from a tool description instead of the user's input, leading to incorrect tool usage. For example, a user credit score of 720 might be replaced by 700 from the tool description. See Appendix H for an example. Both **SPA** and **DPA** showed this behavior, though DPA was less prone due to node-specific tool restrictions.

User Simulator Failures: We observed failures from the LLM based user simulator, which do not reflect agent performance but can affect evaluation reliability. JourneyBench helped identify two issues: **user input hallucination**, where the simulator provides info not in the user seed, and **incomplete user journeys**, where the conversation ends prematurely before the required journey is completed(Appendix I, J).

5 Related Work

Recent literature has explored evaluating LLMs in multi-turn, tool-use settings. Benchmarks

	SPA*				DPA†			
Model	**E-commerce**	**Loan Application**	**Telecommunications**	**Average**	**E-commerce**	**Loan Application**	**Telecommunications**	**Average**
GPT-4o (Wu et al., 2024)	0.617	0.651	0.423	**0.564**	0.730	0.776	0.646	**0.717**
GPT-4o-mini (Wu et al., 2024)	0.502	0.504	0.304	0.437	0.679	0.623	0.646	**0.649**
Claude 3.5 Haiku (Anthropic, 2024)	0.359	0.286	0.116	0.253	0.593	0.615	0.525	0.578
Llama 3.3 (Grattafiori et al., 2024)	0.360	0.278	0.119	0.252	0.432	0.329	0.228	0.330

Table 3: User Journey Coverage Scores (UJCS) for Dynamic-Prompt-Agent (DPA†) and Static-Prompt-Agent (SPA*) across customer service domains. Higher scores indicate better performance.

like Tau Bench (Yao et al., 2024) pioneered simulation-driven evaluation, and its successor Tau2-Bench (Barres et al., 2025) extended this to dual-control environments. Others, like ToolSandBox (Lu et al., 2025) and AppWorld (Trivedi et al., 2024), focus on stateful execution and world-state tracking. While these frameworks advanced agent evaluation, they primarily measure tool selection accuracy or state changes, not an agent's fidelity to a prescribed, multi-step journey with complex dependencies, a gap JourneyBench addresses. For instance, BFCLV3 (Yan et al., 2024) uses static conversations but does not test agent responses to dynamic tool failures. A recent survey by Mohammadi et al. (2025) provides a comprehensive overview of LLM agent evaluation. Our work, in contrast, specifically evaluates policy adherence, task dependencies, and robustness to common conversational disturbances.

Furthermore, a common theme in existing benchmarks is the need to manually define tool logic and database states. This approach poses a challenge to scale to production environments where tools and data constantly evolve. Following principles of separation of concerns from system design, JourneyBench treats tools as modular components with well-defined interfaces, decoupling their internal implementation from workflow evaluation. This design allows our benchmark to focus on an agent's adherence to workflow logic rather than tool-specific behavior, a key distinction from prior work.

6 Conclusion

Moving customer support beyond rigid IVR systems requires agents that combine conversational flexibility with strict policy adherence: a capability existing benchmarks fail to measure. We introduced JourneyBench, a benchmark that evaluates policy-aware agents through graph-based SOP representations and the User Journey Coverage Score metric. Across 703 conversations spanning three domains, we demonstrated that structured workflow orchestration (Dynamic-Prompt-Agent) significantly outperforms prompt-based approaches (Static-Prompt-Agent), enabling even smaller models to exceed larger ones in policy compliance. Our approach is validated in production, where DPA-based systems reliably handle 6,000+ daily customer interactions. By providing both rigorous evaluation methodology and evidence that structured control enables robust, policy-compliant automation, JourneyBench establishes a foundation for deploying reliable AI agents in high-stakes business environments.

7 Limitations and Future Work

Our framework shows strong utility but has limitations that suggest avenues for future research. The **Dynamic-Prompt-Agent**'s success depends on precise modeling of business logic, which can be challenging in dynamic or poorly documented fields. Future work might explore semi-automated graph generation from conversation logs. Our simulation-based evaluation may not capture all nuances of real-world user behavior, and the high cost ($388.88) constrained the range of models we tested. Future research could focus on more cost-effective evaluation methods and complex dependency structures.

8 Ethical Considerations

The use of synthetically generated benchmarks raises important considerations that we address through our methodology and recommend for practitioners.

Synthetic Data Quality: As JourneyBench uses LLMs to generate workflows and conversations, it may inherit model biases. We mitigate this through domain-expert validation (Section 2.2) and QA-based checks of conversational realism (Section 4.2). Organization benchmarks should be paired with real-world bias checks in deployed systems.

Evaluation Validity: LLM-generated evaluation data can introduce circularity. JourneyBench limits this risk by evaluating adherence to human-defined SOP structures rather than free-form generation. Human validation and alignment with production behavior provide additional grounding. We recommend using JourneyBench alongside human assessment.

Workforce Impact: Customer support automation can affect staffing. Our deployments suggest a shift toward higher-complexity tasks rather than direct displacement, but organizations should plan responsible transitions and training.

Acknowledgments

We thank Akshat Bhaskar and Navtej Reddy for their contribution towards implementing production ready versions of this.

References

Anthropic. 2024. Claude 3.5 haiku. `https://www.anthropic.com/claude/haiku`. Accessed: 2025-05-18.

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

Ecaterina Coman. 2025. Ivr systems used in call center management: a scientometric analysis of the literature. *Frontiers in Computer Science*, 7.

CrewAI Inc. 2023. Crewai. `https://github.com/crewAIInc/crewAI`.

D.H. Dean. 2008. What's wrong with ivr self-service. *Managing Service Quality: An International Journal*, 18(6):594–609.

Aaron Grattafiori et al. 2024. The llama 3 herd of models. *Preprint*, arXiv:2407.21783.

LangChain AI. 2023. Langgraph. `https://github.com/langchain-ai/langgraph`.

Jiarui Lu et al. 2025. Toolsandbox: A stateful, conversational, interactive evaluation benchmark for llm tool use capabilities. *Preprint*, arXiv:2408.04682.

Mahmoud Mohammadi, Yipeng Li, Jane Lo, and Wendy Yip. 2025. Evaluation and benchmarking of llm agents: A survey. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2*, KDD '25, pages 6129–6139, New York, NY, USA. Association for Computing Machinery.

Timo Schick et al. 2023. Toolformer: Language models can teach themselves to use tools. *Preprint*, arXiv:2302.04761.

Harsh Trivedi et al. 2024. Appworld: A controllable world of apps and people for benchmarking interactive coding agents. *Preprint*, arXiv:2407.18901.

Xiangyu Wen, Jianyuan Zhong, Zhijian Xu, and Qiang Xu. 2025. Guideline compliance in task-oriented dialogue: The chained prior approach. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 6750–6776, Albuquerque, New Mexico. Association for Computational Linguistics.

Yiqi Wu, Xiaodan Hu, Ziming Fu, Siling Zhou, and Jiangong Li. 2024. Gpt-4o: Visual perception performance of multimodal large language models in piglet activity understanding. *Preprint*, arXiv:2406.09781.

Fanjia Yan et al. 2024. Berkeley function calling leaderboard.

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

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. *Preprint*, arXiv:2210.03629.

A Illustrative Examples for Journey Bench Components

A.1 Example User Seed

Below is an example of a user seed guiding the simulated user:

```
Simulate a conversation to take the
    agent through the following journey.
     Be creative, don't explicitly ask
    for the titles used in the journey
    representation. Follow and trigger
    the sub-steps sequentially. Stop the
     conversation after the final step
    and don't proceed forward. Tell the
    agent that is enough:
1. Initial Application Review
* To trigger Identity Verification
    Provide information: <applicantId>
2. Credit Score Evaluation
* To trigger Credit Report Fetching
    Provide information: <applicantId>
* To trigger Credit Score Analysis
    Provide information: <creditScore>
3. Risk Assessment
* To trigger Risk Evaluation Provide
    information: <financialStatus>
```

A.2 Example Tool Response

An example of a tool's JSON response after successful execution:

```
{
 "https://api.risk.com/assesspost": {
  "success": true,
  "status": "success",
  "message": "Successfully processed
      request for Risk Evaluation",
  "response": {
   "id": "96df4bc8-03d8-4792-92d4-61
       f35a087e1a",
   "timestamp": "2025-05-13T11
       :59:39.539463",
   "tool": "Risk Evaluation",
```

```
    "endpoint": "https://api.risk.com/
        assess",
    "method": "POST",
    "riskLevel": "acceptable"
   }
  }
}
```

A.3 Example Node Definition

A node definition includes its task steps, pathways, and available tools. Below is an example of a "Risk Assessment" node's structure and its associated tools:

```
{
 "id": "5",
 "task_name": "Risk Assessment",
 "task_description": "Perform a
     comprehensive risk assessment to
     determine the feasibility of
     approving a loan based on the
     applicant's financial status.",
 "steps": [
  "Step 1: Utilize the Risk Evaluation
      tool to assess the risks...",
  "Step 2: Analyze the output from the
      Risk Evaluation tool..."
 ],
 "responsePathways": [
  {
   "conditions": [
    {
     "algebraicExpression": "{riskLevel}
          == 'acceptable'"
    }
   ],
   "nextNodeId": "7"
  },
  {
   "conditions": [
    {
     "algebraicExpression": "{riskLevel}
          == 'high'"
    }
   ],
   "nextNodeId": "8"
  }
 ],
 "tools": [
  {
   "method": "POST",
   "url": "https://api.risk.com/assess",
   "body": "{\"applicantId\":\"
       applicant_123\",\"financialStatus
       \":\"financialStatus\"}",
   "name": "Risk Evaluation",
   "tool_description": "Evaluate risks
       associated with the applicant.",
   "condition": null,
   "extractVars": [
    {
     "variableName": "financialStatus",
     "type": "string",
     "description": "financialStatus (
         string): Current financial
         status indicator for risk
         calculation in the Risk
         Evaluation tool. Must be one of
         'Good', 'Fair', 'Poor'."
    }
   ],
   "responseData": [
    {
     "name": "riskLevel",
     "context": "riskLevel (string):
         Indicates risk level. Must be
         one of: 'acceptable', 'high'.
         Example: 'high'."
    }
   ]
  }
 ]
}
```

The key fields in the node definition guide the agent's behavior. `task_description` and `steps` provide natural language instructions. The `tools` array defines the specific APIs the agent can call, including their parameters (`extractVars`) and expected outputs (`responseData`). Crucially, the `responsePathways` (i.e., conditional pathways) encode the procedural logic, defining which `nextNodeId` to transition to based on the result of a tool call. The framework evaluates the `algebraicExpression` at runtime using Python's `eval()` function with the variable values from tool responses, ensuring deterministic transitions.

B Graph Generation Prompt

```
Think about the workflow as a whole
    picture before generating the graph.
     Consider the logical flow of tasks,
     dependencies, and conditions
    required to complete the workflow in
     the domain of {domain_name}. Once
    you have a clear understanding of
    the overall workflow, generate a **
    highly complex graph** with **
    approximately {node_count} nodes**
    for the workflow.

The graph should represent a detailed
    workflow with **multiple pathways**,
     **conditional branching**, and **
    dependencies** between nodes. Each
    node should represent a specific
    task or action in the workflow and
    include attributes such as task name
    , description, steps, tools, and
    response pathways. The graph should
    also include edges representing
    dependencies between nodes, with
    clear labels for each edge.

### Requirements:
1. **Node Count**:
   - The graph must contain **
       approximately {node_count} nodes
       **. Each node should represent a
       unique task or action in the
       workflow.
   - Ensure that the nodes are logically
        connected and represent a
```

```
   complete workflow from start to end.

2. **Graph Connectivity**:
   - The graph must be connected. Every node (except the starting node with `id: 1`) must have at least one incoming edge.
   - Ensure that there are no isolated nodes or subgraphs.
   - For example:
     - If Node 2 exists, it must have at least one edge pointing to it from another node (e.g., Node 1 to Node 2).

3. **No Cycles (Directed Acyclic Graph)**:
   - The graph must not contain any cycles. A cycle occurs when there is a path from a node back to itself (directly or indirectly).
   - For example:
     - Node A to Node B to Node C to Node A (this is a cycle and is not allowed).
   - Ensure that there are no backward edges that create circular dependencies between nodes.
   - The graph must be a **Directed Acyclic Graph (DAG)**, where all edges flow in one direction, and no node can be revisited once it has been processed.

4. **Single End Node**:
   - The graph must have exactly **one end node**. An end node is a node that has no outgoing edges.
   - For example:
     - If Node {node_count} is the end node, it should not have any `responsePathways` or outgoing edges.
   - Ensure that all pathways in the graph eventually lead to this single end node.

5. **Multiple Pathways**:
   - Some nodes must have **multiple outgoing edges** leading to different nodes. These pathways should be based on conditions defined in the `responsePathways` field.
   - Each pathway must have a clear condition (using `algebraicExpression`) that determines which path to follow.
   - For example:
     - If `responseVar1 == 'success'`, go to Node 2.
     - If `responseVar1 == 'failure'`, go to Node 3.
   - Ensure that at least **5 nodes** have multiple outgoing pathways.

6. **Dependencies Between Tools**:
   - Some nodes must include **multiple tools** that are dependent on each other. For example:
     - Tool 1 generates a response variable that is used as input for Tool 2.
     - Tool 2 generates a response variable that is used as input for Tool 3.
   - These dependencies must be explicitly mentioned in the `condition` field of the tools.

7. **Nodes**:
   - Each node should have the following attributes:
     - `id`: A unique identifier for the node.
     - `task_name`: The name of the task.
     - `task_description`: A brief description of the node's function.
     - `steps`: A list of steps that describe the process the task will follow to deliver services. Use indentation to describe sub-steps.
     - `tools`: A list of tools required to perform the task. This can include REST API calls, database queries, or other actions.
       - Each tool should have the following attributes:
         - `method`: The HTTP method to be used (e.g., GET, POST).
         - `url`: The URL for the API call.
         - `condition`: Specifies conditions under which the API call should be made. If there is any dependency on the previous tool, this field should specify the condition.
           - `name`: The name of the condition.
           - `algebraicExpression`: The algebraic expression that defines the condition. This can include logical operators and comparisons.
         - `name`: A name for the action, which can be used for logging or debugging.
         - `tool_description`: A description of the tool's purpose.
         - `extractVars`: A list of variables to give input to the API call. This should include:
           - `variableName`:
               - The name of the variable to give input to the API call.
               - The `variableName` must be unique within the node.
```

```
            - `type`: The type of the
                variable (e.g., string,
                number).
            - `description`:
                - The `description`
                    field should
                    describe the purpose
                     of that variable.
                - The description field
                    should specify what
                    type of values the
                    variable can take or
                     cannot take.
                - If the variable is
                    categorical, you
                    should strictly
                    define the allowed
                    values (name them)
                    in the description
                    field.
        - `responseData`: The response
            of the API call. This
            should include:
            - `name`: The name of the
                response variable.
            - `context`: The context in
                which the variable is
                used.

8. **Response Pathways**:
   - Each node must define `
       responsePathways` to determine
       the next node(s) based on
       conditions.
   - For example:
     - If `responseVar2 == 'valid'`, go
         to Node 6.
     - If `responseVar2 == 'invalid'`,
         go to Node 7.
   - Ensure that at least **5 nodes**
       have multiple `responsePathways`.

9. **Edges**:
   - Each edge should have the following
       attributes:
     - `source`: The ID of the source
         node.
     - `target`: The ID of the target
         node.
     - `label`: A label for the edge,
         which can be used for logging
         or debugging.

10. **Graph**:
    - The graph should have a `title`
        and `description` at the top
        level.
    - The graph should have a `nodes`
        array that contains all the
        nodes in the graph.
    - The graph should have an `edges`
        array that contains all the
        edges in the graph.

### Additional Requirements:
- The graph should include **multiple
    pathways and conditions**, with **
    approximately {node_count} nodes**.
- Ensure that the graph has a clear
    start node and a single end node.
- Include at least **5 nodes with
    conditional pathways** based on API
    responses.
- Ensure that the graph is logically
    consistent and complete.
- Ensure that every node (except the
    starting node with `id: 1`) has at
    least one incoming edge.
- Some nodes must include multiple tools
    , and these tools should depend on
    the results of previous tools within
     the same node. Use the `condition`
    field to specify these dependencies.

### Output:
Think out step by step and generate a
    plan of the graph to generate that
    fits all the requirements. Think out
     the user journeys, tools, response
    pathways and dependencies that need
    to be covered in the generated graph
     and mention it. Aim for high
    quality graphs and realistic
    workflows. Create as detailed as
    needed. Do not over-explain, be
    concise in the amount of text.
```

C Detailed Graph Generation and Validation

During Phase 1 of graph generation (Structure Generation with Synthetic Data), the LLM-crafted foundational graph structures are subjected to a stringent validation process. This process includes:

- **Start Node and Reachability:** Ensuring a single, designated start node from which all other nodes in the graph are accessible.

- **Graph Connectivity:** Confirming that all nodes and edges are correctly linked, with no isolated components.

- **Cycle Detection:** Verifying that the graph is a Directed Acyclic Graph (DAG), thus avoiding infinite loops during navigation.

- **Variable and Expression Validation:** Ensuring all variables used in tool inputs or conditional expressions are well-defined within the graph and that pathway conditions are syntactically correct.

Should any validation fail, the LLM is re-engaged with feedback detailing the issues and suggesting necessary amendments. This iterative refinement, akin to self-correction or reflection methodologies, persists until a valid graph is achieved.

D Algorithm for Condition-Driven Value Generation

To generate synthetic tool responses that ensure specific pathways are taken during user journey generation, the values for variables involved in conditional expressions are determined algorithmically. The process is as follows:

1. **Condition Parsing:** Each conditional expression string (from ‘responsePathways‘ or a tool’s own ‘condition‘ field) is parsed. The system is designed to handle common comparison operators: ‘==‘, ‘>=‘, ‘>‘, ‘<=‘, and ‘<‘. Compound conditions involving logical AND (‘&&‘) and OR (‘||‘) are also supported by breaking them down into their constituent sub-expressions, each of which must resolve to true for the overall path to be considered.

2. **Operator and Value Extraction:** For each sub-expression, we identify the comparison operator and extract the variable name (e.g., ‘{var_name}‘) and the raw value it is compared against.

3. **Type Conversion:** The raw value from the expression is parsed into its likely data type: boolean (‘true’/‘false’), integer, float, or string (stripping enclosing single quotes for string literals).

4. **Value Adjustment for Condition Satisfaction:** Based on the operator and the parsed value, an **adjusted value** is computed for the variable to ensure the sub-expression evaluates to true. This is the crucial step for deterministic path traversal:
   - For ‘==‘: The variable is assigned the parsed value directly (e.g., if condition is `{status} == ’active’`, the synthetic response for ‘status’ will be ’active’; if `{isVerified} == true`, ‘isVerified’ becomes true).
   - For ‘>‘ and ‘>=‘ with numeric types: The variable is assigned ‘parsed_value + 1‘ (e.g., if `{credit_score} >= 720`, ‘credit_score’ is set to 721; if `{count} > 5`, ‘count’ is set to 6).
   - For ‘<‘ and ‘<=‘ with numeric types: The variable is assigned ‘parsed_value - 1‘ (e.g., if `{risk_level} < 3`, ‘risk_level’ is set to ‘2’; if `{attempts} <= 1`, ‘attempts’ is set to 0).

E Static Prompt Agent Template

```
Format Guide:
- Each section represents a node with
    its tools and description. Use only
    the tools listed in the section you
    are in.
- Conditions from the previous node must
     be satisfied before proceeding to
    the next section
- Sections are separated by long lines
    (-----)
- Do not make additional tool calls if
    not explicitly requested by user.
- Keep track of the section you are in
    and the tools available to you. Do
    not mix tools or descriptions from
    different sections.
- After every tool use, communicate the
    result to the user and proceed if
    user requests it.


Following contains a description of the
    node and the logical steps to be
    taken within it. Proceed only if
    requested by the user. Do not
    consider it as an instruction to
    carry out unless user request
    requires it.
Description: Conduct an initial review
    of the applicant's information to
    ensure completeness and validity
    before proceeding with further
    processing.

Steps:
- Step 1: Collect the applicant's data,
    ensuring that all necessary fields
    are populated.
- Step 2: Validate the collected data
    against predefined criteria to
    identify any discrepancies or
    missing information.
- Step 3: Use the Identity Verification
    tool to verify the applicant's
    identity by making an API call with
    the applicantId extracted from the
    collected data. The verify process
    will ensure that the identity status
     is either 'valid' or 'invalid' for
    further action.

Tools:
- Identity Verification

----------------------------------------

If IdentityStatus equals 'valid':
  Then: Below section logic is
      accessible

 Else: Below section logic is not
     accessible

Following contains a description of the
    node and the logical steps to be
    taken within it. Proceed only if
    requested by the user. Do not
    consider it as an instruction to
```

```
    carry out unless user request
    requires it.
Description: This node evaluates an
    applicant's credit score by fetching
     the credit report and analyzing the
     score provided within it.

Steps:
- Step 1: Utilize the Credit Report
    Fetching tool to obtain the
    applicant's credit report by
    providing the applicant's
    alphanumeric ID (applicantId).
    Ensure that the creditReport status
    is either 'available' or '
    unavailable'.
- Step 2: If the creditReport status is
    'available', proceed to analyze the
    credit score using the Credit Score
    Analysis tool. Extract the credit
    score from the fetched report to
    evaluate the credit score status.
- Step 3: If the creditReport status is
    'unavailable', terminate the credit
    score evaluation process and notify
    the applicant about the inability to
     fetch the credit report.

Tools:
- Credit Report Fetching
- Credit Score Analysis (requires Credit
     Report Fetching to be successful
    and response field to meet following
     condition: CreditReport equals '
    available')

----------------------------------------
```

F User Simulation Prompt Template

The following template is used consistently across all experiments to simulate user behavior:

```
Goal:{user_seed}
In each turn of the chat, explicitly
    mention what you want to achieve or
    ask for. The agent will not know
    what you want. You must drive the
    conversation.
Do not repeat information that is
    already provided in the chat. If you
     need to refer to something, you can
     use the context provided in the
    chat.
Give the parameter value listed in the
    seed along with your request in
    every message.
If the agent asks to proceed with a task
     or action after all steps in the
    goal are completed, Strictly say "No
    " and do not proceed with the task.
    End the conversation naturally.

User Information:
Following are the user parameters that
    you can use in your responses:{
    user_info}
If you notice some parameters missing,
    it means you do not have them. DO
    NOT create your own values. Explain
    to the assistant that you do not
    have that information.
Warning: Never provide user information
    that is not present in user
    information section. Do not create
    your own values. Do not collude with
     the assistant to create values.

The user journey can move in one or more
     steps at a time. You can only move
    on to the next task once you have
    completed the current one or if
    agent is unable to finish it due to
    error.

Guidelines:
1. Stay focused on your goal but be
    natural in conversation. Do not veer
     off-topic.
2. Only information available in your
    user seed can be used in the
    conversation. If there are
    parameters missing it means you do
    not have them.
3. If asked for information not
    specified in your context, politely
    mention you don't have that
    information
3. Keep responses concise and to the
    point
4. End the conversation naturally once
    your goal is achieved or if it's
    clear it cannot be achieved.
5. Send a special token "<quit>" after
    you and the assistant have exchanged
     goodbyes at end of conversation. Do
     not send the message along with
    other text. It must be a separate
    message.
6. Ensure the user journey is done
    before sending the "<quit>" token.
    Do not end prematurely.
7. Before sending the "<quit>" token,
    mention all the tasks you were
    supposed to complete to the agent by
     referencing the user seed. Ask the
    agent to confirm that all tasks are
    done. Complete the incomplete tasks
    before sending the "<quit>" token.

Remember: You are simulating a real user
    . Make your responses sound natural
    and human-like.
```

G Dependency Violations

When handling missing parameter and failing function scenarios, our study demonstrates that the Static-Prompt-Agent violates the tool dependency as per the SOP unlike Dynamic Prompt Agent.

G.1 Missing Parameters

```
User: I'm sorry, but I don't have my
    Customer ID with me right now. Can
    we proceed with what we have or
    perhaps move on to the next step?
```

Static Prompt Agent GPT-4o Response:

```
Assistant: No problem at all! We can
    move on to other tasks. If there's
    anything specific you'd like to
    address, such as checking for an
    outage or exploring upgrade options,
     please let me know!
```

Dynamic Prompt Agent GPT-4o-mini Response:

```
Assistant: Unfortunately, I can't
    proceed with the verification
    without specific details such as
    your Customer ID. This ID is
    essential to verify your account.
```

G.2 Failing Function

```
Assistant: It appears that there was an
    error processing your customer
    verification request. Unfortunately,
     I cannot complete the verification
    at this moment due to an internal
    server issue.

User: That's okay. In the meantime,
    could you help me check if there's a
     service outage in my area? My
    customer area is Downtown.
```

Static-Prompt-Agent GPT-4o Response:

```
Assistant: There is currently an outage
    affecting the Downtown area. Is
    there anything else you would like
    assistance with, such as reporting
    this outage or exploring other
    services?
```

Dynamic-Prompt-Agent GPT-4o-mini Response:

```
Assistant: Unfortunately, without
    specific tools to perform this check
    , I'm unable to provide that
    information.
```

H Hallucinations in Parameter Values

The following example, observed in both agent types, demonstrates a hallucination error in parameter values.

In the below conversation, the agent/tool should use the credit score value of 720 as provided by the user. However, as shown, it instead uses the example value 700 from the parameter's description (see Extracted Variable Definition above). This demonstrates a hallucination error where the agent/tool defaults to the example value rather than extracting and using the actual user-supplied value.

Variable Definition:

```
"extractVars": [
    {
        "variableName": "creditScore",
        "type": "integer",
        "description": "creditScore (
            integer): Extracted credit
            score from the report,
            applicable when creditReport
             is 'available'. Example
            value: 700. Invalid: -50, `
            seven hundred'. Must be a
            valid integer."
    }
]
```

Static-Prompt-Agent GPT-4o

```
assistant: Hello! How can I assist you
    today with the loan application
    processing?
user: Hi! I want to get started with the
     loan application process. Can you
    begin with the initial application
    review? My applicant ID is user789.
assistant: Great! Let's start with the
    initial review to ensure everything
    is in order.

I'll first verify your identity using
    the applicant ID provided. Please
    give me a moment.
tool: {
  "id": "ef213b89-cf49-432e-9d17-
      c77857ad3841",
  "timestamp": "2025-05-13T11
      :58:54.694599",
  "tool": "Identity Verification",
  "endpoint": "https://api.example.com/
      identity/validate",
  "method": "POST",
  "identityStatus": "valid"
}
assistant: Your identity has been
    successfully verified, and the
    status is valid.

Would you like me to proceed to the next
     step, which involves evaluating
    your credit score?
user: Yes, please proceed with
    evaluating my credit score. You can
    use my applicant ID, which is
    user789, to fetch the report.
tool: {
  "id": "5a453aae-2621-4867-bc0d-57
      d871abd3e7",
  "timestamp": "2025-05-13T11
      :59:03.233365",
  "tool": "Credit Report Fetching",
  "endpoint": "https://api.credit.com/
      report",
  "method": "GET",
  "creditReport": "available"
}
assistant: Your credit report is
    available. I will now analyze the
    credit score. Please hold on.
tool: {
  "id": "d1e7a30b-1c0a-4735-9621-827
      aa3b50277",
  "timestamp": "2025-05-13T11
      :59:09.838160",
```

```
  "tool": "Credit Score Analysis",
  "endpoint": "https://api.credit.com/
      analyze",
  "method": "POST",
  "creditScoreStatus": "good"
}
assistant: The analysis of your credit
    score indicates that it is 700,
    which is good.

Would you like to continue with a
    comprehensive financial assessment?
```

Dynamic-Prompt-Agent GPT-4o:

```
assistant: Hi there! How can I assist
    you with your loan application today
    ?
user: Hi! I'd like to start with
    understanding where my loan
    application is at. Can we begin with
     the review of my initial
    application? My applicant ID is
    user789.
assistant: To review your initial
    application, I'll need to verify
    your identity as part of the process
    . Let's start by confirming your
    identity using the applicant ID you
    provided.

Please confirm that your applicant ID is
     ``user789'' so I can proceed with
    the verification.
user: Yes, my applicant ID is indeed
    user789. Please proceed with the
    identity verification.
tool: {
  "id": "ef213b89-cf49-432e-9d17-
      c77857ad3841",
  "timestamp": "2025-05-13T11
      :58:54.694599",
  "tool": "Identity Verification",
  "endpoint": "https://api.example.com/
      identity/validate",
  "method": "POST",
  "identityStatus": "valid"
}
assistant: Your identity has been
    successfully verified. Let's proceed
     with reviewing the details of your
    loan application.

What specific information would you like
     to review or verify about your
    application?
user: Great, thank you! I would like to
    evaluate my credit score next. Can
    you fetch the credit report for me
    using my applicant ID, which is
    user789?
tool: {
  "id": "5a453aae-2621-4867-bc0d-57
      d871abd3e7",
  "timestamp": "2025-05-13T11
      :59:03.233365",
  "tool": "Credit Report Fetching",
  "endpoint": "https://api.credit.com/
      report",
  "method": "GET",
  "creditReport": "available"
}
assistant: The credit report for your
    applicant ID is available. Could you
     please provide me with the credit
    score from the report so I can
    analyze it for you?
user: Sure! My credit score is 720.
    Please proceed with the analysis.
tool: {
  "id": "d1e7a30b-1c0a-4735-9621-827
      aa3b50277",
  "timestamp": "2025-05-13T11
      :59:09.838160",
  "tool": "Credit Score Analysis",
  "endpoint": "https://api.credit.com/
      analyze",
  "method": "POST",
  "creditScoreStatus": "good"
}
assistant: Your credit score of 700 has
    been analyzed and is considered ``
    good.'' This should positively
    impact your loan application.
```

I User Input Hallucination Example

The following example illustrates the simulated user hallucinating information it wasn't provided in the user seed

Expected Behaviour: The agent is asking the user for their payment method choice among 'Credit Card', 'PayPal', or 'Bank Transfer'. However, the user was not provided any payment method in the user seed. Hence, the conversation should not have proceeded.

Actual Behaviour: When agent is asking about the payment method choice among 'Credit Card', 'PayPal', or 'Bank Transfer', The user answers credit card as the payment method.

```
assistant: Hello! How can I assist you
    with your shopping or order today?
user: Hi there! I'm looking to start an
    order. Could you help me with that,
    please?
assistant: Great! Could you please
    provide me with the following
    details to get started with your
    order?

1. **Order ID**: It should be an
    alphanumeric string, typically 10-12
     characters long. Example: '
    ORD123456789'.
2. **Payment Method**: Choose from '
    Credit Card', 'PayPal', or 'Bank
    Transfer'.

Once I have these details, I'll help you
     validate and process the order.
user: Sure, here is the order ID:
    AB123C4567D. For now, let's focus on
     validating the order with the given
     order ID. Could you please assist
    me with that?
```

```
assistant: To proceed with the
    validation of your order, I still
    need the payment method you intend
    to use. Could you please provide
    that, selecting from 'Credit Card',
    'PayPal', or 'Bank Transfer'?
user: I'd like to use a Credit Card for
    the payment. Could you now help me
    with the order validation using the
    order ID AB123C4567D?
tool: {
  "id": "b18dc97b-b286-469d-aa6c-
      c65ffdb217dc",
  "timestamp": "2025-05-08T12
      :31:43.703098",
  "tool": "Order Validation Tool",
  "endpoint": "https://api.ecommerce.com
      /orders/validate",
  "method": "POST",
  "isValid": true
}
assistant: Your order ID ``AB123C4567D''
     with the payment method ``Credit
    Card'' has been successfully
    validated!
```

Listing 1: Example of parameter hallucination in user agent

J Incomplete User Journey Example

The following example illustrates a user agent failure where the conversation terminates prematurely, before the prescribed journey is complete.

Expected Behaviour: The full journey required the agent to proceed to 'Service Calls Analysis' after retrieving billing information.

```
Customer Data Verification -> Outage
    Status Check ->... -> Billing Info
    Retrieval -> Service Calls Analysis
```

Actual Behaviour: Here, the user agent ends the conversation after the 'Billing Info Retrieval' tool is called, failing to complete the user journey required.

```
user: Thank you for that information. I
    realize that I wanted to retrieve my
     billing information. Can you assist
     with that? My <customerID> is
    B2345678910.
tool: {
  "id": "61985a25-9a21-4264-8a20-6
      fa9b21f9fa8",
  "timestamp": "2025-05-08T12
      :34:04.482871",
  "tool": "Billing Info Retrieval",
  "endpoint": "https://api.telecom.com/
      getBillingInfo",
  "method": "GET",
  "billingStatus": "`under_review'"
}
assistant: The retrieval of your billing
     information indicates that it's
    currently under review. This means
    there may be some discrepancies
    being checked by the billing
    department.

For further assistance or clarification
    about this review, I recommend
    contacting our billing support team
    directly. They will provide detailed
     information and any necessary next
    steps.

If there's anything else I can assist
    you with, feel free to let me know!
user: No, that should cover everything I
     needed. Thank you for your
    assistance! Just to confirm, here
    are the tasks I aimed to complete.
```

K Human Validation Protocol

We describe the human-in-the-loop review applied to every LLM-generated SOP graph:

- **Reviewer pool:** Five contact center agents (domain experts) independently review each graph.
- **Criteria (binary pass/fail per criterion):**
    1. *Logical Structure:* The flow is logically correct and executable end-to-end; no unreachable nodes; clear start and termination.
    2. *Coherence:* Node and tool descriptions, parameters, and conditional expressions are contextually appropriate and internally consistent.
    3. *Complexity:* Appropriate difficulty for the domain (neither trivial nor needlessly convoluted), with sufficient branching and dependencies.
- **Acceptance:** A graph is included only if all five annotators unanimously pass all three checks.
- **Remediation:** Graphs failing any check are revised (e.g., fix unreachable edges, tighten parameter specifications, clarify tool outcomes) and re-reviewed until passing; otherwise they are excluded.
- **Rationale for binary scoring:** Enables fast, consistent decisions and high inter-annotator agreement; finer-grained scales are left to future work.

Five domain experts independently marked each graph on three yes/no checks (Logical Structure, Coherence, Complexity). A graph is eligible only

Agreement level (graph-level)	Count	Share (%)
5-of-5 on all three checks	4	40.0
≥4-of-5 on all three checks	7	70.0
Total graphs	10	100

Table 4: Experiments use 3 diverse graphs (one per domain) selected from the 4 unanimously validated graphs.

when all five reviewers mark "yes" on all three checks. Out of 10 candidate graphs, 4 met this unanimity requirement. The table below summarizes graph-level agreement 4.

L Realism Validation Summary

We externally assess realism by comparing synthetic conversations to production evaluations on a proprietary dataset of 1,000+ calls. An LLM-as-a-judge rubric is applied with two pillars:

- **Conversational Proficiency (CP):** 5–7 Yes/No questions, e.g., "Was the AI agent's tone consistently kind, empathetic, and friendly?" "Were responses concise and clear?"
- **Goal Attainment (GA):** 5–7 Yes/No questions, e.g., "Did the AI agent accurately identify and capture the caller's intent?" "Was the primary request resolved or properly escalated?"

Each question is scored 1 (Yes) or 0 (No) and averaged per conversation, then averaged across conversations. On synthetic conversations, averages are: Overall 84.37% (CP 82.33%; GA 87.78%). These are comparable to production QA bands obtained with the same rubric used in production QA.

M IVR vs. DPA: Mid-Flow Correction

Grouping dependent tools within a node enables natural user corrections without restarting the process, unlike typical tree-structured IVR flows. The excerpt below illustrates a correction handled mid-node by the Dynamic-Prompt-Agent:

```
assistant: Let's assess eligibility.
    Please share your income category: '
    Salaried', 'Self-Employed', or '
    Unemployed'.
user: Salaried.
assistant: Noted. Proceeding to collect
    your income details.
tool: { "tool": "Income Details
    Collection", "method": "GET", "
    saveStatus": "saved" }
user: Sorry, correction. I'm Self-
    Employed.
assistant: Acknowledged. Updating and re
    -collecting income details.
tool: { "tool": "Income Details
    Collection", "method": "GET", "
    saveStatus": "saved" }
assistant: Proceeding to validate income
    .
tool: { "tool": "Income Validation", "
    method": "POST", "
    incomeValidationResult": "valid" }
assistant: Running financial health
    assessment.
tool: { "tool": "Financial Health
    Assessment", "method": "POST", "
    financialHealthStatus": "positive" }
```