ITERACT-BENCH: BENCHMARKING AGENTS FOR INTERACTIVE DATA ANALYSIS WORKFLOWS

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM-based data agents increasingly support analytical workflows, yet existing benchmarks typically provide requirements upfront or evaluate interaction without testing whether the resulting decisions remain consistent across dependent analyses. In practice, agents must acquire analytical requirements through clarification and feedback, then apply them as persistent constraints on subsequent computations. We introduce ITERACT-BENCH, a benchmark for evaluating these capabilities in interactive data analysis. Built from real-world Kaggle workflows, ITERACT-BENCH contains 86 tasks comprising 554 analytical stages over 727 data files. A user simulator provides specification-guided clarification and correction, while independent executable verifiers assess stage-level correctness without exposing gold outputs to the simulator. We evaluate data agents using seven frontier LLMs and four agent harnesses. The strongest configuration achieves an Average Pass Rate (APR) of 70.9%, leaving substantial room for improvement in analytical reliability. Higher APR is strongly associated with proactive clarification. Agents that seek missing requirements before submitting results tend to perform better, and this behavior varies substantially across models and harnesses. Performance also generally declines as workflows contain more stages and deeper dependencies, highlighting the difficulty of maintaining analytical correctness across complex workflows. ITERACT-BENCH enables systematic evaluation of how agents acquire requirements through interaction and preserve them as constraints on downstream analysis. The data and code are available at https://huggingface.co/datasets/blue-orbit-472/IteractBench.

1 INTRODUCTION

Data analysis extracts useful information from data through cleaning, transformation, and statistical computation, with applications across business intelligence, healthcare, finance, and scientific research (Han et al., 2011). Large language model (LLM)-based data agents make such analysis more accessible by translating natural-language requests into executable analytical workflows (Luo et al., 2026; Zeng et al., 2026). Their adoption is already emerging in practice, with organizations such as Anthropic (Anthropic, 2026b) and OpenAI (Xu et al., 2026a) deploying data-agent systems that enable users to interact with and analyze large-scale organizational data. This growing deployment motivates benchmarks that evaluate data agents under the demands of real-world analytical workflows.

In these workflows, agents must establish what to compute through interaction as they carry out the analysis. Users typically express high-level goals, leaving agents to determine the necessary operations and identify unresolved choices as they inspect data and intermediate results. For example, a high-level request to compare sales channels may require defining eligible orders, computing revenue and repeat-purchase metrics, and combining them into a ranking (Figure 1). Choices such as whether refunded orders count or how competing metrics should be weighted depend on business conventions or user preferences that the data alone cannot determine. Agents must identify which decisions require user input, ask targeted questions, and revise their analysis using user feedback.

Crucially, decisions established through interaction determine what constitutes a correct computation in later stages. An order eligibility rule clarified during cleaning must also govern subsequent revenue and repeat-purchase calculations, whose results feed into the final ranking. An agent may implement the ranking formula correctly yet produce an invalid comparison by recomputing its inputs over

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

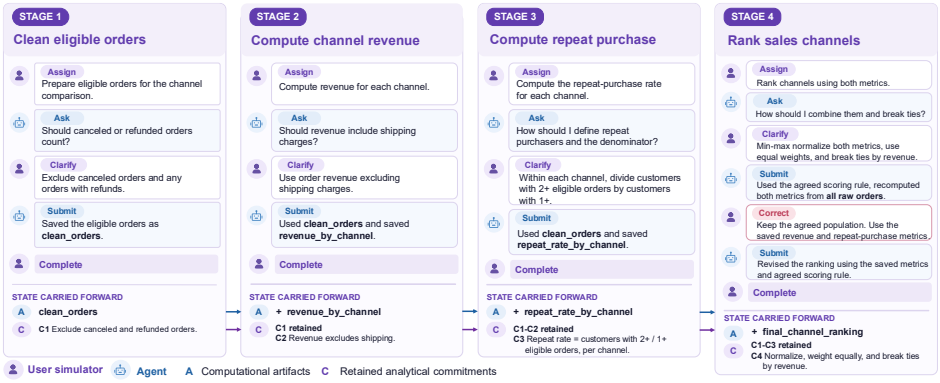


Figure 1: An illustrative interactive analytical workflow in ITERACT-BENCH. Across four stages, the agent interacts with a user simulator to clarify analytical requirements and incorporate corrective feedback. Computational artifacts (A) and retained analytical commitments (C) carry forward to constrain subsequent computations. Stage completion does not imply analytical correctness, which is assessed separately by independent verifiers.

all raw orders. Reliable analysis therefore requires maintaining both *computational artifacts*, such as cleaned tables and intermediate statistics, and *analytical commitments*, such as user-established population definitions and metric conventions. These commitments must remain effective constraints wherever downstream computations depend on them.

Existing benchmarks for data agents often evaluate fully specified tasks, with analytical objectives and requirements supplied upfront (Li et al., 2024a; Egg et al., 2025; Lai et al., 2026; Sahu et al., 2025). Interactive benchmarks introduce clarification and correction, but primarily emphasize local ambiguity resolution or user-directed analytical progression (Li et al., 2024b; Dutta et al., 2026; Li et al., 2025). Whether agents can acquire requirements during execution and preserve them as constraints across dependent analyses remains insufficiently evaluated.

To address this gap, we introduce ITERACT-BENCH, a benchmark for interactive data analysis in which agents progressively acquire analytical requirements and must consistently apply them across dependent stages. Each task comprises a predefined DAG of analytical stages executed in a shared data environment. At each stage, a user simulator issues an *Initial Request*, and the agent determines the operations and additional information needed to fulfill it. Guided by a fixed *Full Task Specification*, the simulator answers clarification questions and provides corrections when user-visible messages expose specification violations. Later stages must follow the requirements clarified with the user and correctly reuse data and intermediate results from earlier stages. Independent executable verifiers compare finalized stage submissions against gold outputs from reference execution, which remain inaccessible to the simulator. Table 1 compares ITERACT-BENCH with existing benchmarks.

We construct ITERACT-BENCH from recent real-world Kaggle analytical workflows through a quality-controlled pipeline combining executable preprocessing, multi-agent construction and review, programmatic validation, and targeted human inspection. The benchmark contains **86 tasks** comprising **554 analytical stages** over **727 data files**, covering **12 analytical stage types**. Tasks average 6.4 stages and contain up to 16; 83.7% include branching or merging dependencies, allowing analytical decisions to constrain multiple downstream computations.

We evaluate data agents on ITERACT-BENCH using seven frontier LLMs and four agent harnesses. The strongest configuration achieves an Average Pass Rate (APR) of **70.9%**, showing that substantial room remains for reliably completing interactive analytical workflows. Our analysis further reveals that a higher APR is strongly associated with proactive clarification. Agents that seek missing requirements before committing to an analysis tend to perform better, while this behavior varies substantially across both models and harnesses. Performance also declines as workflows involve more stages and deeper dependencies, highlighting the challenge of preserving analytical correctness throughout increasingly complex workflows. Together, these results demonstrate how ITERACT-BENCH enables systematic evaluation of whether data agents can acquire necessary requirements through interaction and consistently carry them through dependent analytical workflows.

Table 1: Comparison with representative data-analysis benchmarks. ✓ denotes explicit support, ◇ denotes partial, implicit, or mixed support, and ✗ denotes no support.

Benchmark	# Tasks	Benchmark Grounding		Dependent Workflow		Specification Dynamics			
		Real Workflow	Real Data	Dependent Stages	Persistent State	On-Demand Specification	Clarify	Correct	Cross-Stage Commitments
DABStep (Egg et al., 2025)	450	✓	✗	✗	✗	✗	✗	✗	✗
KramaBench (Lai et al., 2026)	104	✓	◇	✗	✗	✗	✗	✗	✗
InsightBench (Sahu et al., 2025)	100	✗	✗	✗	✗	✗	✗	✗	✗
DAComp (Lei et al., 2026)	236	✗	◇	✗	✗	✗	✗	✗	✗
LongDS-Bench (Xu et al., 2026b)	68	✗	✓	✓	✓	✗	✗	✗	◇
TAPilot-Crossing (Li et al., 2024b)	1,024	◇	◇	✗	◇	◇	◇	✓	✗
ConDABench (Dutta et al., 2026)	1,420	◇	✓	✗	✗	✓	✓	✗	✗
IDA-Bench (Li et al., 2025)	25	✓	✓	◇	◇	✗	◇	✓	◇
ITERACT-BENCH	**86**	✓	✓	✓	✓	✓	✓	✓	✓

Notes. **Real Workflow** indicates that the analytical objective is grounded in a pre-existing real-world analytical workflow rather than being constructed entirely from synthetic task templates. **Real Data** indicates evaluation over real-world datasets. **Dependent Stages** indicates that later user-facing analytical stages explicitly depend on outputs, definitions, or artifacts established in earlier stages. **Persistent State** indicates that analytical state can be retained and reused across stage boundaries. **On-Demand Specification** indicates that answer-affecting requirements are revealed in response to agent-initiated clarification or an observable specification conflict, rather than being proactively supplied as subsequent analytical instructions. **Cross-Stage Commitments** indicates that analytical requirements established through interaction in one stage can constrain the correctness of subsequent dependent stages.

2 BENCHMARK DESIGN

ITERACT-BENCH evaluates data agents in workflows where analytical requirements are progressively acquired through user interaction and must be consistently applied across dependent stages. Its design connects specification-guided user simulation with persistent analytical state and independent executable verification, enabling evaluation of whether agents acquire necessary requirements through interaction, follow them in the current analysis, and preserve them across dependent stages.

2.1 BENCHMARK SETTING

Interactive Analytical Workflow. We model each task in ITERACT-BENCH as an interactive data analysis workflow:

$$\mathcal{T} = (\mathcal{D}, \mathcal{G}, \tau, s_0, \mathcal{U}), \qquad \mathcal{G} = (\mathcal{N}, \mathcal{E}), \tag{1}$$

where $\mathcal{D}$ is the shared data environment, $\mathcal{G}$ is a DAG of analytical stages, τ is a predefined topological execution order, s_0 is the initial analytical state, and $\mathcal{U}$ is the user simulator. Each node represents a user-facing analytical stage with its own objective and structured output:

$$n_i = (\mathcal{D}_i, I_i, F_i, \mathcal{Y}_i, \mathcal{V}_i). \tag{2}$$

Here, $\mathcal{D}_i$ identifies relevant data sources and upstream artifact references, I_i is the *Initial Request*, F_i is the *Full Task Specification*, $\mathcal{Y}_i$ is the structured output space, and $\mathcal{V}_i$ is the stage verifier. Both I_i and F_i are provided to the user simulator $\mathcal{U}$, which uses I_i to introduce the stage and F_i to guide clarification and correction. The agent determines the analytical operations and additional information needed based on its interaction with the simulator. The specification F_i defines the answer-affecting requirements used for evaluation and remains fixed during execution.

Analytical State and Dependencies. We conceptually decompose the evolving analytical state as:

$$s = (A, C), \tag{3}$$

where A contains computational artifacts, including data versions, cleaned tables, and intermediate statistics, while C contains the agent's retained analytical commitments, including population definitions, filtering rules, and metric conventions. Both components reflect the agent's actual execution and may be incomplete or incorrect.

Executing stage i transforms its incoming state s_i^- into an outgoing state s_i^+ and produces candidate submissions in $\mathcal{Y}_i$. An edge $(n_j, n_i) \in \mathcal{E}$ indicates that correctness at stage i depends on designated

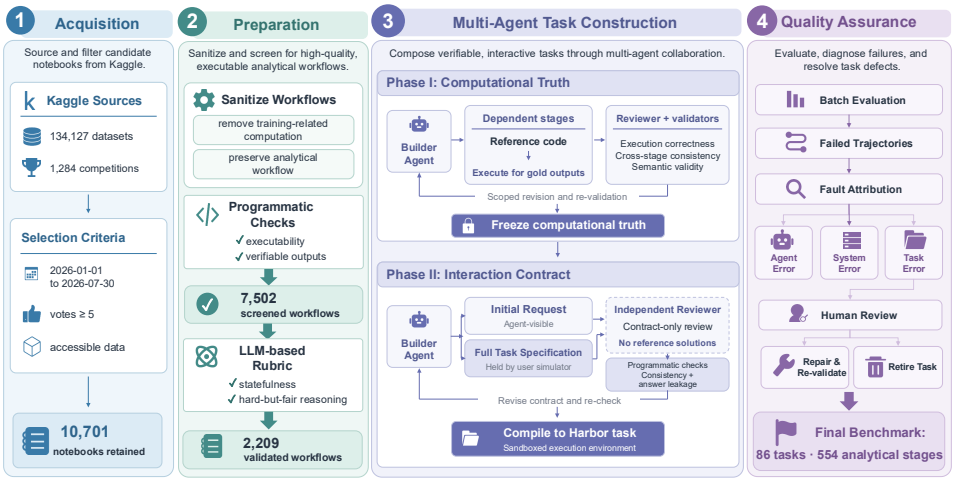


Figure 2: Construction pipeline of ITERACT-BENCH.

artifacts or analytical commitments established in stage j. For example, an order eligibility rule acquired through clarification may constrain later revenue calculations, repeat-purchase statistics, and channel rankings. These dependencies require agents to apply earlier requirements wherever they remain relevant, alongside any newly acquired requirements. Corrections revise the agent's understanding or implementation without changing the underlying specifications. Stages execute according to τ in a persistent environment; the runtime retains the agent's actual state, including errors, rather than replacing it with reference state.

Interaction and Objective Verification. The agent communicates through two actions, ASK and SUBMIT, while the user simulator $\mathcal{U}$ supports four actions: ASSIGN, CLARIFY, CORRECT, and COMPLETE. As illustrated in Figure 1, at the beginning of each stage, the simulator issues ASSIGN with the Initial Request I_i. The agent may use ASK to seek missing information, and the simulator responds with CLARIFY based on the Full Task Specification F_i. Alternatively, the agent may use SUBMIT to provide a structured answer together with an optional explanation. If the submission exposes a specification violation, the simulator responds with CORRECT, allowing the agent to revise its analysis. Clarification and correction keep the stage open. When no remaining observable specification violation is identified, the simulator issues COMPLETE. The runtime then closes the stage and advances to the next stage in the predefined order, or ends the episode. Through these exchanges, requirements are progressively revealed in response to agent questions or observable conflicts, while F_i remains fixed. The simulator has no access to gold outputs or reference implementations, so its acceptance does not establish analytical correctness. When a stage closes, the latest structurally valid submission, if any, is recorded as its finalized answer. After the episode ends, independent executable verifiers compare these answers against gold outputs obtained from reference execution. Detailed disclosure policies and runtime rules, including structural validation, interaction budgets, and termination behavior, are provided in Appendices A.3 and A.4. A worked example in Appendix A.3.1 illustrates how a targeted clarification resolves an answer-affecting ambiguity.

2.2 BENCHMARK CONSTRUCTION

We construct ITERACT-BENCH through a four-step pipeline that transforms real-world analytical workflows into interactive and verifiable agent tasks. Figure 2 provides an overview, with detailed filtering criteria and validation procedures deferred to Appendix B.

Step 1: Acquisition. We collect public Jupyter notebooks and their associated data from Kaggle datasets and competitions (Kaggle, 2026). To reduce potential data contamination from model pre-training, we focus on datasets and competitions created between January 1 and July 30, 2026, and prioritize notebooks with positive community feedback and accessible data. In total, we crawl metadata from *134,127 datasets* and *1,284 competitions*; after filtering for recency, popularity, data availability, and basic execution feasibility, the acquisition stage retains *10,701 notebooks*.

Step 2: Preparation. Real-world notebooks often mix analytical workflows with model development, exploratory trials, and other components unsuitable for our evaluation setting. We therefore sanitize each candidate by removing training-related computation while preserving the analytical workflow required for subsequent task construction. Programmatic checks further screen candidates for executability, analysis depth, and basic verifiability, leaving *7,502 workflows*. We then apply an LLM-based rubric to assess properties that require semantic judgment, such as stateful analytical dependencies, hard-but-fair reasoning, and exact verifiability. Together, the programmatic and semantic filters yield *2,209 validated analytical workflows* for task construction.

Step 3: Multi-Agent Task Construction. We formulate task construction as an iterative multi-agent workflow with two phases. In the first phase, a Builder Agent decomposes a validated workflow into dependent analytical stages, constructs reference implementations, and executes them to obtain gold outputs. Programmatic validators and an independent Reviewer Agent check execution correctness, cross-stage consistency, and semantic validity; failed checks trigger scoped revision and re-validation before the *computational truth* is frozen. In the second phase, the Builder constructs the interaction contract for each stage, including its Initial Request and Full Task Specification. A separate Reviewer Agent independently examines the contract without access to reference solutions, while programmatic checks guard against inconsistency and answer leakage. This Builder–Reviewer loop continues until the task passes both computational and interaction-level validation, after which it is compiled into a runnable Harbor task in a sandboxed environment.

Step 4: Quality Assurance. Finally, we run the constructed tasks with multiple agent configurations and perform fault attribution on failed trajectories, distinguishing *agent*, *execution-system*, and *task* errors. Suspected task defects receive targeted human review: recoverable specification issues are repaired and re-validated without modifying the frozen gold, while irrecoverably ambiguous tasks are removed. Before finalization, every task undergoes human review to assess specification clarity, unintended answer leakage, and analytical difficulty, with low-quality tasks removed. This process results in the final benchmark of **86 tasks** comprising **554 analytical stages**.

2.3 Benchmark Statistics and Characteristics

Figure 3 summarizes the scale, domain coverage, and workflow structure of ITERACT-BENCH. The benchmark contains 86 tasks comprising 554 analytical stages and 727 data files, covering 12 analytical stage types. As shown in Figure 3(a), tasks span a broad range of application domains. Sports and competitions account for the largest share (24.4%), followed by education and labor (15.1%), with the remaining tasks distributed across other domains.

The workflows exhibit substantial variation in length and dependency structure. Tasks contain 2–16 stages, averaging 6.4, while DAG depth ranges from 2 to 8, averaging 3.2. Overall, 80.9% of stages depend on upstream analytical states, and 83.7% of tasks contain non-chain DAG structures with branching or merging dependencies. These dependencies require agents to correctly reuse earlier data transformations, intermediate results, and analytical decisions in subsequent computations.

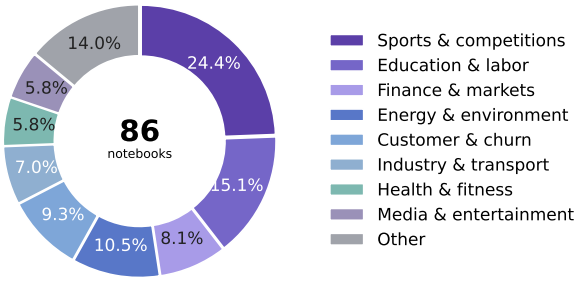
Within these dependent workflows, agents must acquire requirements that are not fully specified in the initial requests through clarification and feedback. The resulting decisions can constrain downstream stages alongside previously produced data and artifacts. Thus, ITERACT-BENCH evaluates whether agents acquire necessary information and consistently apply it across dependent stages.

3 Experiments

3.1 Setup

Evaluation Protocol. We evaluate all systems on ITERACT-BENCH, comprising 86 tasks and 554 analytical stages, with three valid trials per model–harness configuration. Stages follow the predefined analytical order, while the runtime enforces interaction budgets and termination rules. Details on trial validity, runtime limits, and execution settings appear in Appendix C.2 and C.3.

Models and Agent Frameworks. We evaluate 7 frontier LLMs: Claude Opus 5 (Anthropic, 2026a), GPT-5.6 Sol (OpenAI, 2026), Kimi-K3 (Team et al., 2026), GLM-5.2 (GLM-5-Team et al., 2026),



Statistic	Value
Scale	
Tasks	86
Analytical stages	554
Data files	727
Stage types	12
Workflow Dependency	
Stages per task	2 / 6.4 / 16 (min / avg / max)
DAG depth	2 / 3.2 / 8 (min / avg / max)
Stages with upstream dependencies	80.9%
Non-chain DAG structures	83.7%

Figure 3: Overview of ITERACT-BENCH characteristics. (a) Subject distribution of the 86 benchmark tasks. (b) Benchmark statistics covering scale and workflow dependencies.

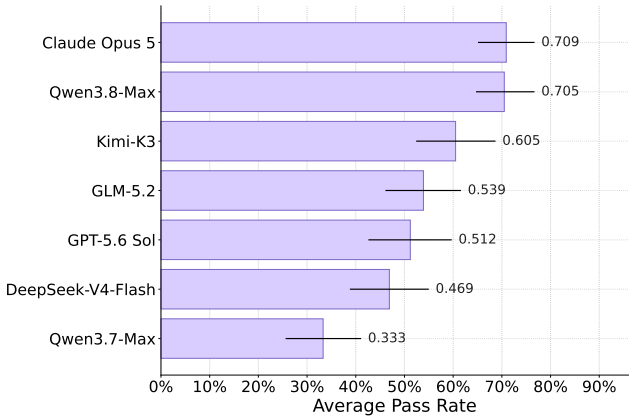


Figure 4: **Model performance under a fixed Pi-Agent harness.** APR of different models evaluated with Pi-Agent, with 95% confidence intervals.

Qwen3.7-Max (Qwen Team, 2026a), Qwen3.8-Max (Qwen Team, 2026b), and DeepSeek-V4-Flash-0731 (DeepSeek-AI, 2026), under four agent frameworks: Pi-Agent (Pi Contributors, 2026), Claude Code (Anthropic, 2026), Codex (OpenAI, 2026), and OpenClaw (OpenClaw Foundation, 2026).

All harnesses interact with ITERACT-BENCH through the same MCP-based interaction interface. They have identical access to benchmark files and retain their default analysis tools, allowing us to evaluate both model capability and agent framework effects under a unified environment. Detailed harness implementations and versions are summarized in Appendix C.1.

Metrics. We use Average Pass Rate (APR) as the primary evaluation metric. A trial is successful only when all analytical stages in a task pass exact verification. For each task, APR averages success outcomes over valid trials, and the final benchmark score is averaged across tasks. The complete metric definition and handling of invalid trials are provided in Appendix C.3.

3.2 MAIN RESULTS

Finding 1. Both model and harness choice substantially affect performance on ITERACT-BENCH. No harness performs best for every model, and even the strongest evaluated configuration achieves only 70.9% APR.

Figure 4 compares seven models under the same Pi-Agent harness. Claude Opus 5 and Qwen3.8-Max achieve the highest Average Pass Rates (APRs), at 70.9% and 70.5%, respectively, with substantially overlapping 95% confidence intervals. Kimi-K3 follows at 60.5%, while the remaining models achieve 33.3%–53.9%. These results reveal substantial performance differences across models under a fixed harness, with considerable room for improvement even for the strongest configuration in

Table 2: APR across model–harness configurations on ITERACT-BENCH.

Model	Pi	OpenClaw	Codex	Claude Code	Model Mean
Qwen3.8-Max	**0.705**	0.508	0.682	0.585	**0.620**
GLM-5.2	0.539	0.264	0.407	**0.570**	0.445
DeepSeek-V4-Flash	**0.469**	0.353	0.426	0.419	0.417
Qwen3.7-Max	**0.333**	0.298	0.279	0.271	0.296
Harness Mean	**0.512**	0.356	0.449	0.461	

correctly completing all stages of a task. Among the four models evaluated across all four harnesses in Table 2, Qwen3.8-Max consistently performs best, achieving a model mean of 62.0%.

Harness choice also substantially affects performance. Across the four models in Table 2, Pi-Agent achieves the highest mean APR at 51.2%, followed by Claude Code at 46.1%, Codex at 44.9%, and OpenClaw at 35.6%. The difference between the highest and lowest harness means is 15.6 percentage points. For individual models, the variation can be even larger: GLM-5.2 achieves 57.0% with Claude Code but only 26.4% with OpenClaw, a gap of 30.6 percentage points. Thus, performance on ITERACT-BENCH depends strongly on how the underlying model is integrated into an agent.

The best harness also varies by model. Pi-Agent performs best for three of the four models, whereas GLM-5.2 achieves its highest APR with Claude Code, exceeding its Pi-Agent result by 3.1 percentage points. Other harness rankings also change across models: Codex outperforms Claude Code for Qwen3.8-Max and DeepSeek-V4-Flash, but the ordering reverses for GLM-5.2.

3.3 IN-DEPTH ANALYSIS

3.3.1 EFFECT OF SPECIFICATION AVAILABILITY

Finding 2. Providing complete specifications upfront substantially improves performance, indicating that ITERACT-BENCH poses additional challenges in acquiring requirements through interaction beyond executing the analysis itself.

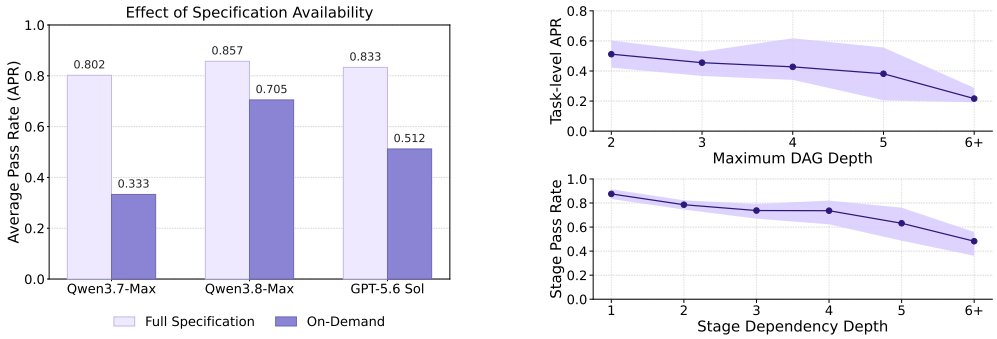
To examine the challenges introduced by interactive specification acquisition, we compare two settings under the same Pi-Agent harness. In *Full Specification*, the agent receives the complete specification at the beginning of each stage. In *On-Demand Specification*, the agent must obtain requirements omitted from the initial request through clarification and feedback from the user simulator.

As shown in Figure 5a, all three models perform relatively well when complete specifications are available, achieving APRs between 80.2% and 85.7%. Performance drops substantially when requirements must instead be acquired through interaction: Qwen3.7-Max falls from 80.2% to 33.3%, GPT-5.6 Sol from 83.3% to 51.2%, and Qwen3.8-Max from 85.7% to 70.5%. These declines indicate that the difficulty of ITERACT-BENCH extends beyond analytical execution to recognizing unresolved requirements, eliciting the necessary information, and applying user feedback.

Moreover, the performance gap between the strongest and weakest models widens from 5.4 to 37.2 percentage points when interaction is required. Models that perform similarly with complete specifications thus differ substantially in their ability to obtain and use those specifications through interaction. This contrast highlights the importance of evaluating interaction capabilities explicitly: strong performance on fully specified analyses alone does not establish reliability in interactive analytical workflows. Additional ablations show that models benefit unevenly from clarification, while allowing both clarification and correction outperforms either channel alone for both tested models (Appendix D.1).

3.3.2 IMPACT OF WORKFLOW COMPLEXITY

Finding 3. Performance declines with dependency depth at both the task and stage levels, highlighting the difficulty of maintaining analytical correctness across dependent computations.



(a) Full Specification vs. On-Demand Specification.

(b) APR under increasing workflow complexity.

Figure 5: (a) Complete specification access versus interactive acquisition. (b) Performance by workflow complexity across all model–harness configurations. Shaded regions denote pointwise 95% confidence intervals.

We analyze performance by dependency depth across all evaluated model–harness configurations. Task-level APR is grouped by maximum DAG depth, with tasks and configurations weighted equally. Stage pass rate pools verification outcomes across configurations and trials by each stage's dependency depth, measured as the number of stages on the longest dependency path ending at that stage; root stages have depth one. All stages are included in evaluation, with unreached stages scored as failures. Shaded regions show pointwise 95% task-level bootstrap confidence intervals, as detailed in Appendix C.3.

Figure 5b shows that task-level APR decreases from approximately 51% at maximum DAG depth two to 22% at depth six or greater. Stage pass rate likewise decreases overall, from approximately 88% at depth one to 48% at depth six or greater. These patterns are consistent with the challenge of preserving upstream artifacts and analytical commitments across dependent computations, although the grouped analysis does not isolate dependency depth from workflow length or stage difficulty. Additional results by workflow length appear in Appendix D.2.

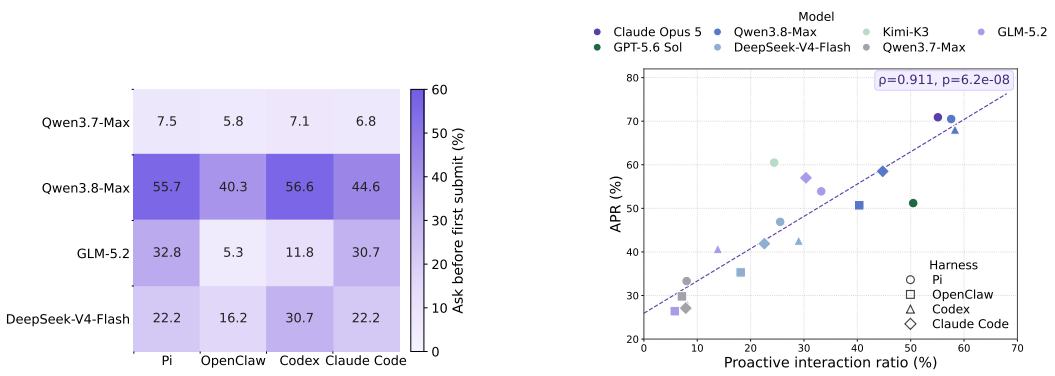
3.3.3 Interaction Behavior

Finding 4. Proactive clarification is associated with higher accuracy, both across configurations and across repeated trials of the same stage, model, and harness.

We first examine whether agents seek clarification before committing to an answer. Figure 6(a) reports the fraction of stages in which an agent asks at least one question before its first submission. This rate varies substantially across model–harness configurations, ranging from 5.3% to 56.6%. Qwen3.8-Max asks proactively in 49.3% of stages on average across harnesses, compared with only 6.8% for Qwen3.7-Max. Harness choice also affects the same model's behavior: GLM-5.2 asks before submission in 32.8% of stages with Pi-Agent but only 5.3% with OpenClaw. Thus, both the model and harness influence whether agents seek missing requirements before submitting an analysis.

To relate interaction behavior to performance, we define the *proactive interaction ratio* as the fraction of clarification events among all clarification and correction events. Figure 6(b) shows a strong positive association between this ratio and APR (Spearman $\rho = 0.911$, $p = 6.2 \times 10^{-8}$). Configurations whose interactions consist more often of clarification rather than correction tend to achieve higher analytical accuracy. This pattern suggests that acquiring requirements proactively is an important component of effective interactive analysis.

To account for differences across configurations, we compare repeated trials of the same stage under the same model and harness. Trials that ask before their first submission show 20.1 percentage points higher first-submission accuracy and 12.3 points higher final-stage accuracy (Appendix D.3). Although observational, this comparison supports the association between proactive clarification and correctness after holding the stage and configuration fixed.



(a) Ask-before-first-submit rate

(b) Proactive interaction ratio vs. APR

Figure 6: **Interaction behavior across model–harness configurations.** (a) The tendency to ask for clarification before the first submission. (b) Configurations with a larger proportion of proactive rather than reactive interaction achieve higher APR.

3.4 Additional Analyses

Additional analyses in the appendix further characterize failure modes and evaluation robustness. Fault attribution shows that *Requirement Acquisition* is a major error source whose prevalence varies substantially with both model and harness choice; prompt inspection provides a plausible explanation for part of the harness-dependent behavior. We also find that unresolved upstream errors can propagate across dependent stages, with weaker models exhibiting both more frequent and longer failure cascades (Appendix D.4). Finally, varying the user simulator shows similar APR across sufficiently capable simulators, while a weaker simulator introduces more simulator-attributed failures and depresses measured performance (Appendix D.5).

4 Related Work

Data Analysis Agent Benchmarks. Existing benchmarks evaluate data agents on analytical reasoning, querying, or insight generation, typically with requirements provided upfront (Egg et al., 2025; Lai et al., 2026; Sahu et al., 2025; Lei et al., 2026). LongDS-Bench (Xu et al., 2026b) examines state maintenance across dependent analyses, but does not explicitly evaluate requirement acquisition through interaction. Interactive benchmarks narrow this gap but retain different simplifications. TAPilot-Crossing (Li et al., 2024b) and ConDABench (Dutta et al., 2026) evaluate clarification or correction within individual analytical tasks, while IDA-Bench (Li et al., 2025) supports multi-round analysis but relies on user-provided instructions. In contrast, IterAct-Bench evaluates requirement acquisition and preservation through interaction (Table 1).

Interactive Agent Benchmarks. Interactive evaluation beyond data analysis covers service-domain assistance in τ-bench and τ^2-bench (Barres et al., 2025), stateful tool use in ToolSandbox (Lu et al., 2025), and social interaction in SOTOPIA (Zhou et al., 2024). UserBench (Qian et al., 2025) and VitaBench (He et al., 2026) further examine clarification and collaboration under underspecified goals and evolving user needs. These benchmarks highlight the importance of interaction but do not specifically evaluate data analysis workflows, where user input defines computational requirements that must remain consistent across dependent stages. IterAct-Bench tests whether agents acquire requirements through interaction and consistently apply them across dependent stages.

5 Conclusion

We introduced IterAct-Bench, a benchmark for interactive data analysis workflows that evaluates whether data agents can acquire analytical requirements through interaction and preserve them across dependent stages. Our experiments show that current agents remain limited by insufficient proactive clarification and difficulty maintaining correctness across dependencies, with performance shaped by both model and harness design. We hope IterAct-Bench supports progress toward more reliable interactive data agents for complex analytical workflows.

AI USE STATEMENT

Generative AI tools were used in this work both as components of the research methodology and as research and writing assistants. As described in the paper, LLMs are used within the benchmark construction, review, user-simulation, and agent-evaluation pipelines; these uses constitute part of the proposed methodology and are evaluated or controlled by the corresponding programmatic checks, executable verifiers, and quality-assurance procedures.

During the research process, we additionally used generative AI tools to assist with refining experimental methodology and analysis design, developing and debugging data-processing and evaluation code, analyzing and interpreting experimental results, and creating or refining scientific figures. We also used generative AI tools for literature search and summarization, brainstorming, and drafting, restructuring, and editing portions of the manuscript for clarity and readability.

All AI-assisted research decisions, analyses, code, figures, and manuscript text were reviewed by the authors. Experimental results reported in the paper were obtained from the actual benchmark executions rather than generated by an LLM, and quantitative claims were checked against the corresponding experimental outputs. The underlying real-world datasets and execution-derived gold outputs were not replaced by LLM-generated data. References and literature-related claims were checked against their original sources. The authors take full responsibility for the methodology, experiments, claims, and final content of the paper.

ETHICS STATEMENT

This work does not involve experiments with human participants. ITERACT-BENCH is constructed from publicly available data-analysis workflows and datasets collected from Kaggle. We use these resources solely for research and benchmark construction and release benchmark artifacts in accordance with the applicable licenses and terms of the original sources.

The benchmark evaluates LLM-based agents and uses LLMs as user simulators. As with other model-based evaluation frameworks, simulator behavior may reflect biases or limitations of the underlying models. We therefore isolate gold outputs from the simulator, use independent executable verification for objective correctness, and report additional analyses of simulator-related errors and evaluation sensitivity.

ITERACT-BENCH is intended for research on reliable interactive data analysis. The benchmark may contain analytical tasks derived from real-world datasets, and conclusions drawn from benchmark performance should not be interpreted as evidence that an agent is suitable for deployment in high-stakes domains without additional domain-specific validation.

REPRODUCIBILITY STATEMENT

We release the benchmark artifacts, including task specifications and executable verifiers, at https://huggingface.co/datasets/blue-orbit-472/IteractBench, and the runtime and evaluation code at https://anonymous.4open.science/r/Iteract-Bench-CDFD. Section 2.1 defines the benchmark setting and evaluation protocol, while the appendix provides details on benchmark construction, user simulation, runtime interaction, verification, and experimental settings. Together, these resources support reproduction of the reported benchmark results.

REFERENCES

Anthropic. Introducing claude opus 5, July 2026a. URL https://www.anthropic.com/news/claude-opus-5. Anthropic Product Announcement.

Anthropic. How anthropic enables self-service data analytics with claude. https://claude.com/blog/how-anthropic-enables-self-service-data-analytics-with-claude, 2026b. Accessed: 2026-09-20.

Anthropic. Claude code. https://github.com/anthropics/claude-code, 2026. Software repository, accessed: 2026-09-21.

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. τ^2-Bench: Evaluating Conversational Agents in a Dual-Control Environment. *arXiv preprint arXiv:2506.07982*, 2025.

DeepSeek-AI. Deepseek-v4: Towards highly efficient million-token context intelligence, 2026.

Avik Dutta, Priyanshu Gupta, Hosein Hasanbeig, Rahul Pratap Singh, Harshit Nigam, Sumit Gulwani, Arjun Radhakrishna, Gustavo Soares, and Ashish Tiwari. Condabench: Interactive evaluation of language models for data analysis. In *Companion of the International Conference on Management of Data*, pp. 281–292, 2026.

Alex Egg, Martin Iglesias Goyanes, Friso Kingma, Andreu Mora, Leandro von Werra, and Thomas Wolf. Dabstep: Data agent benchmark for multi-step reasoning. *arXiv preprint arXiv:2506.23719*, 2025.

GLM-5-Team, :, Aohan Zeng, Xin Lv, Zhenyu Hou, Zhengxiao Du, Qinkai Zheng, Bin Chen, Da Yin, Chendi Ge, Chenghua Huang, Chengxing Xie, Chenzheng Zhu, Congfeng Yin, Cunxiang Wang, Gengzheng Pan, Hao Zeng, Haoke Zhang, Haoran Wang, Huilong Chen, Jiajie Zhang, Jian Jiao, Jiaqi Guo, Jingsen Wang, Jingzhao Du, Jinzhu Wu, Kedong Wang, Lei Li, Lin Fan, Lucen Zhong, Mingdao Liu, Mingming Zhao, Pengfan Du, Qian Dong, Rui Lu, Shuang-Li, Shulin Cao, Song Liu, Ting Jiang, Xiaodong Chen, Xiaohan Zhang, Xuancheng Huang, Xuezhen Dong, Yabo Xu, Yao Wei, Yifan An, Yilin Niu, Yitong Zhu, Yuanhao Wen, Yukuo Cen, Yushi Bai, Zhongpei Qiao, Zihan Wang, Zikang Wang, Zilin Zhu, Ziqiang Liu, Zixuan Li, Bojie Wang, Bosi Wen, Can Huang, Changpeng Cai, Chao Yu, Chen Li, Chengwei Hu, Chenhui Zhang, Dan Zhang, Daoyan Lin, Dayong Yang, Di Wang, Ding Ai, Erle Zhu, Fangzhou Yi, Feiyu Chen, Guohong Wen, Hailong Sun, Haisha Zhao, Haiyi Hu, Hanchen Zhang, Hanrui Liu, Hanyu Zhang, Hao Peng, Hao Tai, Haobo Zhang, He Liu, Hongwei Wang, Hongxi Yan, Hongyu Ge, Huan Liu, Huanpeng Chu, Jia'ni Zhao, Jiachen Wang, Jiajing Zhao, Jiamin Ren, Jiapeng Wang, Jiaxin Zhang, Jiayi Gui, Jiayue Zhao, Jijie Li, Jing An, Jing Li, Jingwei Yuan, Jinhua Du, Jinxin Liu, Junkai Zhi, Junwen Duan, Kaiyue Zhou, Kangjian Wei, Ke Wang, Keyun Luo, Laiqiang Zhang, Leigang Sha, Liang Xu, Lindong Wu, Lintao Ding, Lu Chen, Minghao Li, Nianyi Lin, Pan Ta, Qiang Zou, Rongjun Song, Ruiqi Yang, Shangqing Tu, Shangtong Yang, Shaoxiang Wu, Shengyan Zhang, Shijie Li, Shuang Li, Shuyi Fan, Wei Qin, Wei Tian, Weining Zhang, Wenbo Yu, Wenjie Liang, Xiang Kuang, Xiangmeng Cheng, Xiangyang Li, Xiaoquan Yan, Xiaowei Hu, Xiaoying Ling, Xing Fan, Xingye Xia, Xinyuan Zhang, Xinze Zhang, Xirui Pan, Xu Zou, Xunkai Zhang, Yadi Liu, Yandong Wu, Yanfu Li, Yidong Wang, Yifan Zhu, Yijun Tan, Yilin Zhou, Yiming Pan, Ying Zhang, Yinpei Su, Yipeng Geng, Yong Yan, Yonglin Tan, Yuean Bi, Yuhan Shen, Yuhao Yang, Yujiang Li, Yunan Liu, Yunqing Wang, Yuntao Li, Yurong Wu, Yutao Zhang, Yuxi Duan, Yuxuan Zhang, Zezhen Liu, Zhengtao Jiang, Zhenhe Yan, Zheyu Zhang, Zhixiang Wei, Zhuo Chen, Zhuoer Feng, Zijun Yao, Ziwei Chai, Ziyuan Wang, Zuzhou Zhang, Bin Xu, Minlie Huang, Hongning Wang, Juanzi Li, Yuxiao Dong, and Jie Tang. Glm-5: from vibe coding to agentic engineering, 2026. URL https://arxiv.org/abs/2602.15763.

Jiawei Han, Micheline Kamber, and Jian Pei. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, San Francisco, CA, USA, 3rd edition, 2011. ISBN 0123814790, 9780123814791.

Wei He, Yueqing Sun, Hongyan Hao, Xueyuan Hao, Zhikang Xia, Qi Gu, Hui Su, and Xunliang Cai. Vitabench: Benchmarking llm agents with versatile interactive tasks in real-world applications. In *International Conference on Learning Representations*, volume 2026, pp. 91654–91681, 2026.

Kaggle. Kaggle. https://www.kaggle.com/, 2026. Accessed: 2026-09-05.

Eugenie Lai, Gerardo Vitagliano, Ziyu Zhang, Om Chabra, Sivaprasad Sudhir, Anna Zeng, Anton Zabreyko, Chenning Li, Ferdi Kossmann, Jialin Ding, et al. Kramabench: A benchmark for ai systems on data-to-insight pipelines over data lakes. In *International Conference on Learning Representations*, volume 2026, pp. 142883–142912, 2026.

Fangyu Lei, Jinxiang Meng, Yiming Huang, Yitong Zhang, Jianwen Luo, Xin Zou, Ruiyi Yang, Wenbo Shi, Yan Gao, Shizhu He, et al. Dacomp: Benchmarking data agents across the full data intelligence lifecycle. In *International Conference on Learning Representations*, volume 2026, pp. 104463–104501, 2026.

Hanyu Li, Haoyu Liu, Tingyu Zhu, Tianyu Guo, Zeyu Zheng, Xiaotie Deng, and Michael I Jordan. Ida-bench: Evaluating llms on interactive guided data analysis. *arXiv preprint arXiv:2505.18223*, 2025.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, 36, 2024a.

Jinyang Li, Nan Huo, Yan Gao, Jiayi Shi, Yingxiu Zhao, Ge Qu, Yurong Wu, Chenhao Ma, Jian-Guang Lou, and Reynold Cheng. Tapilot-crossing: Benchmarking and evolving llms towards interactive data analysis agents. *arXiv preprint arXiv:2403.05307*, 2024b.

Jiarui Lu, Thomas Holleis, Yizhe Zhang, Bernhard Aumayer, Feng Nan, Haoping Bai, Shuang Ma, Shen Ma, Mengyu Li, Guoli Yin, et al. Toolsandbox: A stateful, conversational, interactive evaluation benchmark for llm tool use capabilities. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 1160–1183, 2025.

Yuyu Luo, Guoliang Li, Ju Fan, and Nan Tang. Data agents: Levels, state of the art, and open problems. In *Companion of the International Conference on Management of Data*, pp. 571–579, 2026.

OpenAI. Codex. https://github.com/openai/codex, 2026. Software repository, accessed: 2026-09-21.

OpenAI. Previewing gpt-5.6 sol: A next-generation model, June 2026. URL https://openai.com/index/previewing-gpt-5-6-sol/. OpenAI Product Announcement.

OpenClaw Foundation. Openclaw. https://github.com/openclaw/openclaw, 2026. Software repository, accessed: 2026-09-21.

Pi Contributors. Pi: A minimal terminal coding harness. https://pi.dev/, 2026. Accessed: 2026-09-21.

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

Qwen Team. Qwen3.7: The agent frontier, May 2026a. URL https://qwen.ai/blog?id=qwen3.7.

Qwen Team. Qwen3.8-max: A new bar for coding and cowork, August 2026b. URL https://qwen.ai/blog?id=qwen3.8.

Gaurav Sahu, Abhay Puri, Juan A Rodriguez, Amirhossein Abaskohi, Mohammad Chegini, Alexandre Drouin, Perouz Taslakian, Valentina Zantedeschi, Alexandre Lacoste, David Vazquez, et al. Insightbench: Evaluating business analytics agents through multi-step insight generation. In *International Conference on Learning Representations*, volume 2025, pp. 4683–4715, 2025.

Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, Jianfeng Cai, Xinyuan Cai, Peizhou Cao, Yuxuan Cao, Ziwei Chai, Y Charles, et al. Kimi k3: Open frontier intelligence. *arXiv preprint arXiv:2607.24653*, 2026.

Bonnie Xu, Aravind Suresh, and Emma Tang. Inside openai's in-house data agent. https://openai.com/index/inside-our-in-house-data-agent/, 2026a. Accessed: 2026-09-20.

Kewei Xu, Xiaoben Lu, Shuofei Qiao, Zihan Ding, Haoming Xu, Lei Liang, and Ningyu Zhang. Longds-bench: On the failure of long-horizon agentic data analysis. *arXiv preprint arXiv:2605.30434*, 2026b.

Tianjing Zeng, Yuntao Hong, Zhongjun Ding, Dandan Liu, Yinan Mei, Yunxiang Su, Yiming Wang, Xiaojian Zhang, Jingyu Zhu, Junhao Zhu, et al. Qwenpaw-data: Bridging facts, methodology, and execution for autonomous enterprise data analytics. *arXiv preprint arXiv:2607.11019*, 2026.

Xuhui Zhou, Hao Zhu, Leena Mathur, Ruohong Zhang, Haofei Yu, Zhengyang Qi, Louis-Philippe Morency, Yonatan Bisk, Daniel Fried, Graham Neubig, et al. Sotopia: Interactive evaluation for social intelligence in language agents. In *International Conference on Learning Representations*, volume 2024, pp. 40975–41019, 2024.

A BENCHMARK DETAILS

A.1 BENCHMARK STATISTICS

We provide additional statistics of ITERACT-BENCH to characterize its scale, workflow complexity, analytical diversity, and interaction requirements. The benchmark contains 86 tasks with 554 analytical stages and 727 associated data files. These statistics demonstrate that ITERACT-BENCH involves multi-stage, dependency-rich analytical workflows rather than isolated single-step analysis.

Stage Distribution. We first analyze the number of analytical stages per task. As shown in Figure 7, tasks contain between 2 and 16 stages, with a mean of 6.4 stages per task. The distribution covers both relatively concise workflows and long-horizon analytical processes: 55.8% of tasks contain at least five stages, while 23.3% contain ten or more stages. This indicates that multi-stage analytical workflows are a common property of ITERACT-BENCH rather than being introduced only by a few outlier tasks.

Dependency Structure. Beyond stage counts, we analyze the dependency structures among analytical stages. A task is considered *chain-only* if all stages form a linear execution path without branching or merging dependencies. As shown in Figure 7(b), only 16.3% of tasks are chain-only, while 83.7% contain non-chain dependency structures. Specifically, 43.0% of tasks involve branching dependencies, 3.5% involve merging dependencies, and 37.2% contain both branching and merging patterns.

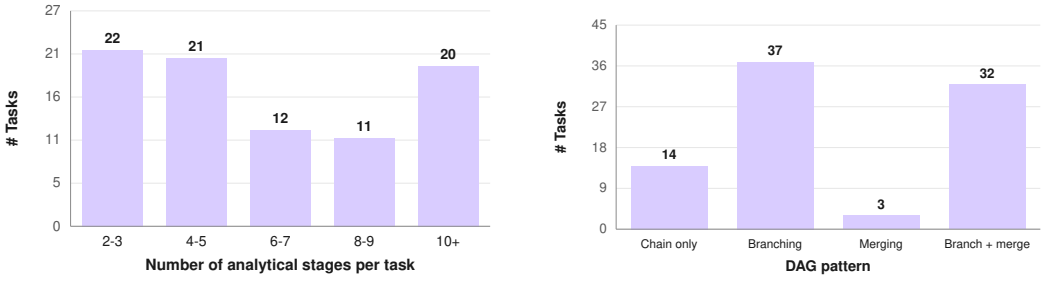
Figure 8 further shows the distribution of maximum DAG depth. The average DAG depth is 3.2, with a maximum depth of 8. Each task contains an average of 5.7 dependency edges, indicating that analytical states frequently propagate across multiple dependent stages.

Analytical Stage Diversity. To characterize the diversity of analytical operations, we categorize each analytical stage into one of 12 operation types. Figure 8 summarizes the distribution.

Statistical analysis is the most frequent category (170 stages), followed by data profiling (88), data transformation (60), representation construction (58), and scoring and ranking (57). The benchmark also contains stages involving data cleaning, integration, validation, retrieval, reporting, and modeling. This diversity shows that ITERACT-BENCH evaluates agents across a broad range of analytical procedures rather than a single query or aggregation pattern.

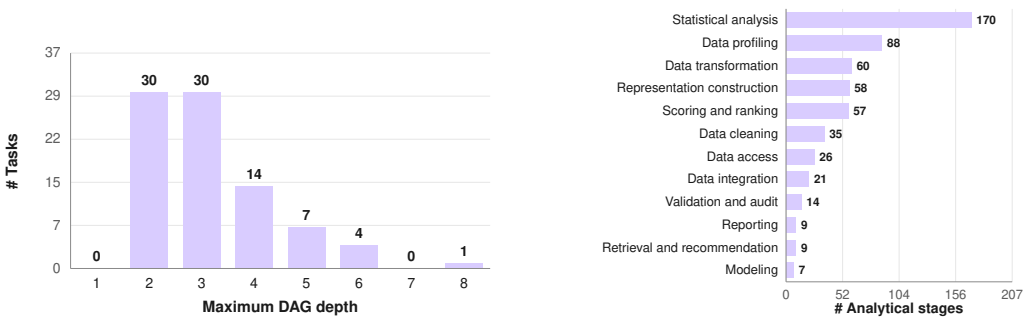
Interactive Requirement Taxonomy. To characterize the interactive specification requirements in ITERACT-BENCH, we analyze the hidden requirements that are intentionally omitted from the initial request and revealed through interaction. Across 554 stages, we identify 2,418 hidden requirements, averaging 4.4 requirements per stage.

Table 3 summarizes the requirement distribution. Metric definition and output specification are the two most frequent categories, accounting for 28.4% and 25.7% of requirements, respectively.



(a) Stage distribution (b) DAG patterns

Figure 7: Workflow complexity statistics of ITERACT-BENCH. (a) Distribution of analytical stages per task. (b) Distribution of dependency patterns, including chain-only, branching, merging, and branch–merge workflows.



(a) DAG depth distribution (b) Analytical stage types

Figure 8: Additional benchmark characteristics. (a) Distribution of maximum DAG depth per task. (b) Distribution of analytical operation types across benchmark stages.

Table 3: Taxonomy of hidden requirements in ITERACT-BENCH.

Requirement Type	Percentage
Metric definition	28.4%
Output specification	25.7%
Population definition	21.3%
Filtering criteria	12.7%
Grouping and comparison	6.1%
Cross-stage dependency	5.7%

Population definition and filtering criteria together account for 34.0% of requirements, reflecting the importance of analytical scope and data selection decisions. Cross-stage dependency requirements capture cases where later stages depend on intermediate analytical states produced earlier in the workflow.

A.2 TASK AND STATE FORMALIZATION

Detailed Task Definition. Following Section 2.1, we represent each task in ITERACT-BENCH as an interactive analytical workflow

$$\mathcal{T} = (\mathcal{D}, \mathcal{G}, \tau, s_0, \mathcal{U}), \qquad \mathcal{G} = (\mathcal{N}, \mathcal{E}),$$

where $\mathcal{D}$ is the shared data environment, $\mathcal{G}$ is a directed acyclic graph (DAG) of analytical stages, τ is a predefined topological execution order, s_0 is the initial analytical state, and $\mathcal{U}$ is the user simulator.

An analytical *stage* is a user-facing analytical task with its own objective and structured output, rather than an internal reasoning or execution step. Each stage $n_i \in \mathcal{N}$ is represented as

$$n_i = (\mathcal{D}_i, I_i, F_i, \mathcal{Y}_i, \mathcal{V}_i),$$

where $\mathcal{D}_i$ identifies the data sources and upstream artifact references relevant to the stage, I_i is the *Initial Request*, F_i is the fixed *Full Task Specification*, $\mathcal{Y}_i$ defines the structured output space, and $\mathcal{V}_i$ is the executable verifier for the stage.

The Initial Request I_i contains the information naturally available when the stage is assigned, whereas F_i contains all answer-affecting requirements needed to determine the intended analysis. Both are available to the user simulator, but the agent is initially exposed only to I_i. During execution, additional information from F_i may be revealed through CLARIFY or CORRECT. Thus, the underlying specification remains fixed throughout the stage, while the agent's knowledge of that specification may progressively expand. Interaction budgets and other runtime constraints are treated as execution parameters rather than part of the semantic stage definition.

Cross-Stage Dependencies. For a stage n_i, let

$$\mathrm{pa}(i) = \{j \mid (n_j, n_i) \in \mathcal{E}\}$$

denote its parent stages. An edge $(n_j, n_i) \in \mathcal{E}$ indicates that correctness at stage i depends on designated computational artifacts or analytical commitments established during stage j.

Such dependencies may require a later stage to reuse an earlier cleaned dataset, intermediate statistic, population definition, metric convention, or other analytical decision. For example, an eligibility rule established while cleaning orders may constrain both subsequent revenue and repeat-purchase calculations, whose outputs are then reused by a final channel-ranking stage. The stages therefore cannot be treated as independent questions even though each has its own user-facing request and output.

Stages are executed according to the fixed topological order τ, which guarantees that all parent stages precede their dependents. The DAG specifies *which* earlier analytical results or commitments are required for downstream correctness; it does not imply that the runtime discards state from non-parent stages. Because execution occurs in a persistent environment, artifacts and other agent-created state from earlier stages may remain available throughout the episode.

Analytical State. We conceptually decompose the evolving analytical state as

$$s = (A, C),$$

where A denotes computational artifacts and C denotes retained analytical commitments.

Computational artifacts include data versions, cleaned or joined tables, filtered datasets, intermediate statistics, and other reusable outputs. Analytical commitments include population definitions, identifier mappings, filtering and aggregation rules, metric definitions, ranking conventions, and tie-breaking policies established or adopted during execution. Both components describe the agent's actual working state and may therefore be incomplete or incorrect.

This decomposition is conceptual and does not require the agent to maintain an explicit structured state object. Artifacts may be persisted in the shared execution environment, while analytical commitments may be reflected in dialogue context, analysis code, or agent-maintained notes. The agent is responsible for retaining and consistently applying whichever parts remain relevant to downstream stages.

Let s_i^- and s_i^+ denote the analytical state immediately before and after stage i, respectively, and let h_i denote the stage trajectory, including user-agent exchanges, tool executions, generated artifacts, corrections, and submissions. We write the resulting transition abstractly as

$$s_i^+ = \Phi_i(s_i^-, \mathcal{D}_i, h_i),$$

where Φ_i represents the state changes induced by the agent's execution and interaction during stage i. Under the persistent runtime, s_i^+ is carried forward and contributes to the incoming state of later stages according to the execution order τ.

Importantly, the runtime propagates the agent's *actual* state rather than replacing it with reference state. A clarification or correction communicates information from the fixed specification F_i, but does not itself repair the agent's artifacts or internal commitments. The agent must incorporate the feedback and revise affected computations when necessary. If it fails to do so, stale commitments or incorrect artifacts may persist and affect downstream analyses.

A.3 User Simulator

The user simulator assigns analytical stages, answers clarification questions, provides corrections, and signals stage completion. It communicates task requirements without access to gold outputs, reference executions, or stage verifiers.

Information Boundary. For each stage i, both the Initial Request I_i and the fixed Full Task Specification F_i are available to the user simulator. However, the agent is initially exposed only to I_i: the simulator uses I_i to generate the initial ASSIGN message, while requirements contained only in F_i remain undisclosed until they become relevant through clarification or correction.

During subsequent interaction, the simulator receives the current stage specification F_i and the MCP interaction history. For submissions, the runtime may additionally provide sanitized structural

Information source	User simulator access
Initial Request I_i	Available for the current stage and used to generate ASSIGN.
Current Full Task Specification F_i	Privately available to guide clarification and correction; requirements not contained in I_i are not disclosed upfront.
MCP interaction history	Available across the task, including agent questions, submissions, and simulator responses.
Submitted methods and answers	Available as communicated, including explanations outside the answer envelope and structured JSON values.
Structural diagnostics	Private, sanitized validation information supplied by the runtime.
Agent code, execution outputs, and console logs	Unavailable unless explicitly included in an MCP message.
File metadata	Available through optional inspection tools: paths, file sizes, and CSV column names and row counts; no cell contents.
Completed-stage summaries	Private summaries for conversational continuity; not exposed to the agent.
Future-stage specifications	Unavailable before stage activation.
Gold outputs, reference executions, and verifier results	Unavailable throughout interaction.

Table 4: Information available to the user simulator. The simulator can observe submitted answer values and the fixed specification for the active stage, but not gold outputs, reference executions, or unexposed agent execution traces.

diagnostics. The simulator does not observe the agent's unexposed code, execution traces, or console outputs.

Table 4 summarizes the simulator's access. Gold outputs and reference executions are reserved for independent verification after task execution and are never available to the simulator during interaction.

Simulator Actions. The user simulator supports four actions:

$$\mathcal{A}_U = \{\text{ASSIGN}, \text{CLARIFY}, \text{CORRECT}, \text{COMPLETE}\}.$$

ASSIGN presents the Initial Request I_i for the stage selected by the runtime. CLARIFY answers an agent question according to the disclosure policy below. CORRECT communicates an observable conflict with the Full Task Specification F_i, or a structural problem reported through sanitized runtime diagnostics. COMPLETE signals that no remaining observable specification violation has been identified and requests stage closure. The first three actions communicate task information to the agent, whereas COMPLETE controls stage completion through the runtime.

Agent Actions. The agent communicates with the simulator through two actions:

$$\mathcal{A}_A = \{\text{ASK}, \text{SUBMIT}\}.$$

ASK and SUBMIT correspond to the MCP tools `ask_question` and `submit_response`, respectively. ASK requests information without submitting an answer, whereas SUBMIT provides a structured answer with an optional explanation. This separation makes clarification requests and candidate submissions explicit in the recorded interaction trajectory. Ordinary console output is not treated as communication with the user.

On-Demand Specification Disclosure. We write the fixed Full Task Specification of stage i as

$$F_i = \{r_i^1, r_i^2, \ldots, r_i^{K_i}\},$$

where each r_i^k is an answer-affecting requirement. The complete set F_i is fixed before stage execution. The Initial Request I_i exposes the subset of information naturally available when the stage is assigned, while additional requirements may subsequently be communicated through CLARIFY or CORRECT.

Our main evaluation uses on-demand disclosure. For a CLARIFY response, the simulator is instructed to address only requirements explicitly queried or assumed in the agent's current message. It must not volunteer other requirements merely because they concern the same analytical stage. Responses paraphrase only the constraints needed to address the question, without reproducing private specification text, requirement identifiers, or internal specification fields. An observable conflict with F_i may instead trigger CORRECT, which communicates the corresponding requirement.

For generic questions such as "Are there any other analysis requirements?", the simulator is instructed not to enumerate all remaining undisclosed requirements. These disclosure restrictions are implemented through simulator prompt instructions rather than a separate deterministic disclosure filter.

Observable Correction. A semantic correction must be grounded in an observable conflict between the fixed specification F_i and information exposed by the agent in the interaction. Such evidence may include a stated population definition, filtering rule, metric convention, ranking rule, or structured answer field that conflicts with F_i. For structural errors, feedback may instead be based on sanitized diagnostics supplied by the runtime.

Method explanations are optional in the main evaluation; no separate mandatory method-review step is imposed. When an agent voluntarily includes a method description in a submission, that description may support correction if it exposes a conflict with F_i. The simulator is instructed not to request additional methodological detail as a prerequisite for completion and not to infer a specification violation merely because a numerical result appears implausible.

The simulator cannot inspect unexposed code or execution traces. A submitted method description therefore reflects what the agent claims to have done rather than independently verified execution. Without such a description, correction remains possible when the structured answer itself exposes a specification violation.

Normal Stage Completion. The simulator issues COMPLETE when no remaining observable specification violation is identified from the available interaction. This action is implemented by `complete_current_turn`, a tool available only to the user simulator.

The runtime accepts COMPLETE only when at least one structurally valid candidate submission exists. It records the latest structurally valid submission as the finalized answer for the stage and advances to the next stage in the predefined execution order τ. If no stages remain, the episode terminates. A natural-language acceptance without an explicit COMPLETE action does not close the stage.

Importantly, COMPLETE reflects only the absence of an observable specification violation; it does not establish objective analytical correctness. After the episode ends, the independent executable verifier $\mathcal{V}_i$ evaluates the finalized stage answer against the gold output obtained from reference execution. Stage-budget exhaustion and task-level termination are handled separately by the runtime, as detailed in Appendix A.4.

Cross-Stage Interaction. Stages execute according to the predefined topological order τ, which respects the dependency DAG $\mathcal{G}$. The runtime determines the next stage, and the simulator introduces it through ASSIGN; the simulator neither selects the execution order nor invents additional analytical stages.

Later stages may depend on computational artifacts A and analytical commitments C established during earlier stages. For example, an order-population definition clarified in one stage may continue to constrain later revenue, repeat-purchase, and ranking computations. The agent is responsible for retaining relevant commitments, incorporating corrections into its working state, and revising affected computational artifacts when necessary. The simulator does not automatically repair or replace the agent's state with reference state.

A.3.1 WORKED EXAMPLE OF ON-DEMAND DISCLOSURE

Figure 9 presents the interaction for Stage 12 of the e-cigarette retail-panel task. The agent uses `cleaned_panel` from Stage 2 to rank brands by their persistence in the top quartile of monthly market share.

Stage 12: Brand persistence — Recorded interaction

```
event 98 | USER SIMULATOR | ASSIGN
Which brands stayed on top, rather than briefly spiking?

Using cleaned_panel, compute each brand's monthly share of national
    unit sales. Identify months in which its share reaches or exceeds
    that month's 75th percentile of brand shares.

For each brand, report the proportion of months in the top quartile,
    peak share, first and last observed shares, and the change
    between those endpoints. Rank by persistence, breaking ties by
    brand name.

Return valid JSON in one <answer> block. Ask for clarification
    before submitting if needed.

event 99 | AGENT | ASK
Six decisions need to be pinned down:

1. Output: should the result be an array under "brands"? What are
    the exact field names? Should first/last month labels be included?

2. Percentile population: use all ten brands every month, filling
    unobserved brands with zero shares, or only brands observed that
    month?

   I obtain 53 top-quartile months for NJOY with zero-filling,
    versus 29 with observed brands only.

3. Persistence denominator: all 68 panel months, or each brand's own
    observed months?

   Geek Bar:     15/68 = 0.2206, or 15/44 = 0.3409.
   Breeze Smoke:  2/68 = 0.0294, or  2/44 = 0.0455.

4. Units: should market shares be percentages or fractions? Should
    all values use four decimal places?

5. Endpoint change: final minus initial share in percentage points,
    or relative percentage change?

6. Other conventions: use each brand's first/last observed month,
    peak share over its observed months, a linearly interpolated
    percentile, an inclusive >= threshold, and national totals
    without row filtering?

event 100 | USER SIMULATOR | CLARIFY
1. Return {"brands": [...]} with exactly these fields:
   brand_name, pct_months_top, peak_share,
   start_share, final_share, share_change.
   Include every brand; omit month labels.
   Sort by pct_months_top descending, then brand_name ascending.

2. Use only brands observed in each month.
   Do not zero-fill unobserved brands.
   Use linear interpolation and an inclusive >= threshold.

3. Use each brand's own observed month count.
   pct_months_top is the fraction of those months that qualify.

4. Market shares use the 0-100 percentage scale.
```

```
   Sum national units across jurisdictions, flavors, and product
    types without row filtering.
   Round at the output boundary to four decimal places.

5. share_change = final_share - start_share.
   Use unrounded shares for the subtraction.
   Report the signed difference in percentage points.

6. Use each brand's first and last observed month.
   peak_share covers all its observed months, including months
    outside the top quartile.

I am not validating the counts or rankings you mentioned. Apply
    these conventions and report the results.

event 101 | AGENT | ASK
One remaining scale question: pct_months_top is described as a
    fraction, but its name begins with "pct_".

For Geek Bar's reported 15 qualifying months out of 44, should I
    submit:
(a) 0.3409 on the 0-1 scale; or
(b) 34.0909 on the 0-100 scale?

I have implemented the other conventions and will submit after this
    confirmation.

event 102 | USER SIMULATOR | CLARIFY
Use (a): pct_months_top is a fraction on the 0-1 scale.
For 15 out of 44, that is 0.3409 to four decimal places.

The 0-100 scale applies to the market-share fields. Do not multiply
    persistence by 100 because of its field name.

Submit when ready. I am not confirming the underlying counts or
    brand rankings.

event 103 | AGENT | SUBMIT
[Two of the ten submitted brand objects are shown.]

{
  "brand_name": "Geek Bar",
  "pct_months_top": 0.3409,
  "peak_share": 12.6267,
  "start_share": 2.6125,
  "final_share": 12.6267,
  "share_change": 10.0142
},
{
  "brand_name": "Breeze Smoke",
  "pct_months_top": 0.0455,
  "peak_share": 8.849,
  "start_share": 1.9699,
  "final_share": 8.849,
  "share_change": 6.8791
}

event 105 | USER SIMULATOR | COMPLETE
complete_current_turn({})

events 106-107 | RUNTIME | STAGE CLOSED
{
  "all_closed": true,
```

```
  "close_reason": "completed",
  "closed_turn_id": 12,
  "next_turn_id": null
}
```

Figure 9: Stage 12 interaction, abridged and paraphrased from trace events 98–107. Event order and reported numerical values are preserved. The submission excerpt retains two complete brand objects; the answer envelope and remaining eight objects are omitted. Counts and alternative calculations originate from the agent's question. The closing tool call records stage completion, not an independent correctness verdict.

The Full Task Specification defines persistence over each brand's own observed months, a denominator left unspecified in the initial request. For the counts reported by the agent, this choice changes Geek Bar's persistence from $15/68 = 0.2206$ to $15/44 = 0.3409$. The first clarification resolves the observation window; the second confirms that persistence is reported as a fraction, despite the `pct_` field prefix. The submitted value follows both conventions.

A.4 Runtime Protocol and Verification

MCP-Mediated Communication. All explicit communication between the agent and the user simulator is mediated by the benchmark runtime through the Model Context Protocol (MCP). The agent is exposed to two interaction tools: `ask_question` and `submit_response`, corresponding to ASK and SUBMIT, respectively. Ordinary console output is not treated as communication with the user.

Communication follows a blocking, half-duplex protocol: at most one user–agent exchange is pending at any time. When the agent invokes an interaction tool, the runtime synchronously invokes the user simulator and returns its response before agent execution continues. This explicit protocol makes the intent and ordering of interaction events unambiguous and supports exchange-level analysis. Figure 10 summarizes the runtime flow.

Stage Lifecycle and Interaction. Stages execute according to the predefined topological order τ. When stage i is activated, the user simulator issues ASSIGN with the Initial Request I_i. The runtime determines which stage is activated next; the user simulator does not select or reorder analytical stages.

During an active stage, the agent may invoke `ask_question` to request missing information. The runtime forwards the question to the user simulator, which has access to the fixed Full Task Specification F_i and responds with CLARIFY according to the disclosure policy described in Appendix A.3. Clarification leaves the current stage open and does not produce an answer submission.

Alternatively, the agent may invoke `submit_response`. Each submission must contain exactly one structured answer envelope,

```
<answer>
{ ... JSON ... }
</answer>
```

whose content is valid JSON and conforms to the stage output schema $\mathcal{Y}_i$. Text outside the envelope may contain user-visible explanation but is not included in objective answer verification.

The runtime performs structural validation only to determine whether a usable structured candidate has been submitted. If a submission is malformed or violates $\mathcal{Y}_i$, the user simulator receives a sanitized structural diagnostic and may return CORRECT. Such structural failures do not invoke the objective verifier and do not erase an earlier structurally valid submission.

Semantic Feedback and Stage Completion. For a structurally valid submission, the user simulator observes the submitted message together with the fixed specification F_i. It may issue CORRECT when the interaction exposes an observable conflict with F_i, such as an incorrect population definition, filtering rule, metric convention, ranking rule, or output requirement. The simulator neither needs nor has access to gold outputs or reference execution when generating this feedback.

The simulator issues the private COMPLETE action when no remaining observable specification violation is identified. This signal is handled internally by the runtime and is not exposed as an agent action; a natural-language statement such as "looks good" therefore does not close the stage.

Upon normal completion, the runtime records the latest structurally valid submission as the finalized answer for stage i and advances according to τ. Importantly, COMPLETE indicates only that the simulator detects no remaining observable specification violation; it does not establish objective analytical correctness.

Per-Stage Interaction Budget. Each stage has a finite interaction budget, with a default of 10 exchanges. Both ASK and SUBMIT consume one exchange, where an exchange consists of an agent interaction action and the corresponding user-simulator response. Structurally invalid submissions also consume this budget.

If the budget is exhausted before normal completion, the runtime closes the stage automatically. The latest structurally valid submission, if one exists, is recorded as the stage answer even without an explicit COMPLETE. If no structurally valid submission exists, the stage is recorded with an empty answer. The runtime then advances according to τ.

This fallback prevents one unresolved stage from blocking the remainder of the workflow while ensuring that unresolved or invalid answers are penalized during objective verification.

Early Task Termination. Task-level termination conditions are separate from the per-stage interaction budget. A task may terminate early if the agent stops execution or the runtime reaches an overall execution or error limit.

Finalized answers from previously closed stages are preserved. For an active stage that has not yet closed, no additional finalization is performed at task termination, and stages that were never reached contain no solution. All stages remain part of the trial success criterion, so early termination does not remove unfinished stages from evaluation.

Gold Isolation and Objective Verification. Objective correctness is deliberately separated from the interaction loop. During interaction, neither the user simulator nor the interaction runtime accesses gold outputs, reference executions, or verifier results. Gold outputs are obtained by executing the frozen reference workflow over the benchmark data and are reserved for independent verification after the episode ends.

Let $\hat{y}_{t,j,i}$ denote the finalized structured answer for stage i of task t in trial j, with $\hat{y}_{t,j,i} = \perp$ when no valid finalized answer exists, and let $y^*_{t,i}$ denote the corresponding execution-derived gold output. Stage-level correctness is

$$r_{t,j,i} = \begin{cases} \mathcal{V}_i(\hat{y}_{t,j,i}, y^*_{t,i}), & \hat{y}_{t,j,i} \neq \perp, \\ 0, & \hat{y}_{t,j,i} = \perp. \end{cases}$$

Each $\mathcal{V}_i$ is deterministic. By default, verification recursively compares the complete JSON structure: required and unexpected fields, scalar values, and list elements must agree with the gold output. Object-key ordering is ignored, whereas list ordering remains significant. Selected numerical fields may specify small explicit tolerances in the task schema; otherwise values are compared exactly up to numerical type equivalence. No LLM judge is used for objective verification.

Although verification is performed at the stage level, benchmark success is defined over the complete analytical workflow. A trial succeeds only when every stage of the task is correct:

$$s_{t,j} = \begin{cases} 1, & r_{t,j,i} = 1,\ \forall n_i \in \mathcal{N}_t, \\ 0, & \text{otherwise.} \end{cases}$$

Thus, partial completion does not constitute task success.

For each task t, we compute the pass rate over K independent trials:

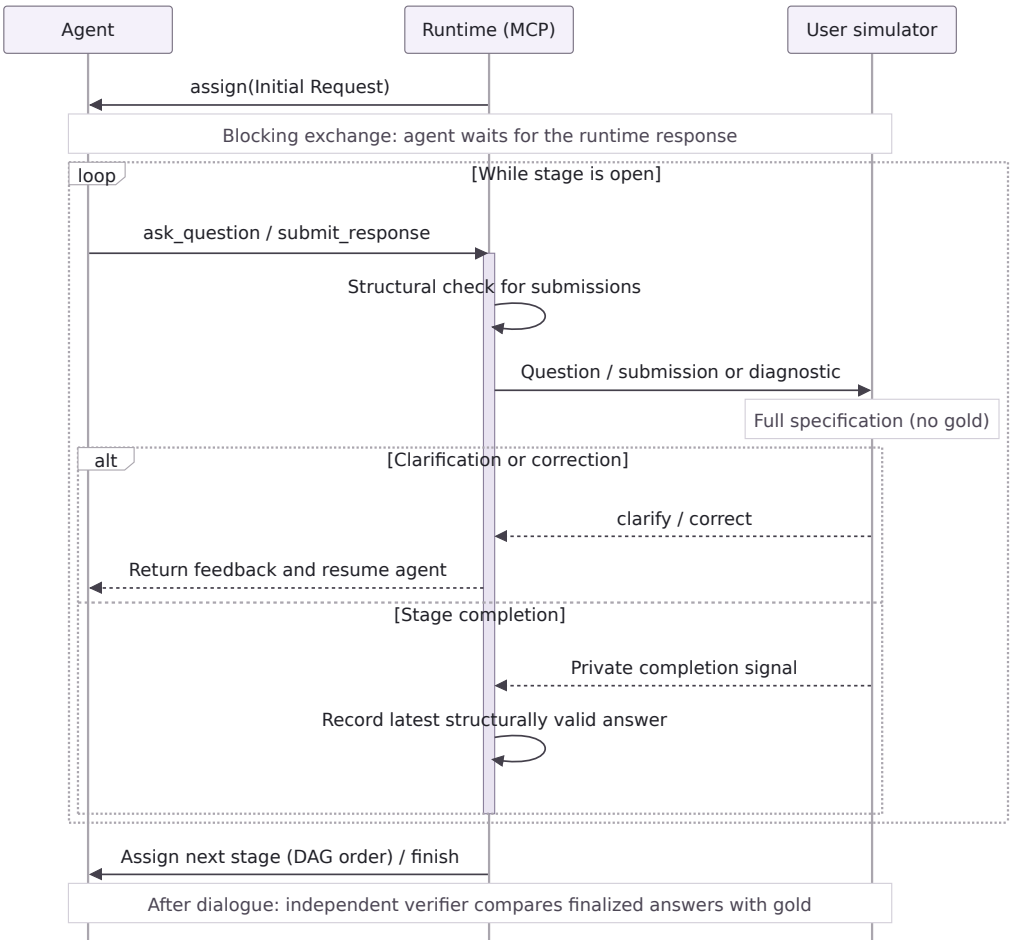


Figure 10: Half-duplex runtime protocol. Agent–user communication is mediated by the runtime through explicit `ask_question` and `submit_response` actions. The runtime performs structural validation only, while gold-based correctness evaluation is isolated from the dialogue.

$$P_t = \frac{1}{K} \sum_{j=1}^{K} s_{t,j}. \tag{4}$$

In our experiments, we use $K = 3$ independent trials per task. The final benchmark metric, Average Pass Rate (APR), is

$$\mathrm{APR} = \frac{1}{|\mathcal{T}|} \sum_{t \in \mathcal{T}} P_t. \tag{5}$$

This task-level aggregation gives each workflow equal weight regardless of its number of analytical stages, while requiring successful trials to satisfy the complete dependent analytical workflow.

B Benchmark Construction Details

This section provides additional details of the construction pipeline introduced in Section 2.2. Our pipeline separates four concerns: (1) acquiring recent real-world analytical workflows, (2) filtering them for executable and verifiable analysis, (3) constructing computationally grounded interactive

tasks through a multi-agent procedure, and (4) performing post-construction quality assurance. A central design principle throughout the pipeline is to separate properties that can be checked deterministically from those that require semantic judgment.

B.1 ACQUISITION

Sources and Temporal Filtering. We collect public Jupyter notebooks and their associated data from Kaggle datasets and competitions (Kaggle, 2026). To reduce the likelihood that benchmark workflows were observed during model pre-training, we restrict both source datasets and competitions to those whose `CreationDate` falls between January 1 and July 30, 2026. The temporal filter is applied to the underlying dataset or competition, rather than to the creation or last-execution time of the notebook itself.

We crawl metadata from **134,127 datasets** and **1,284 competitions**. Within the retained sources, we consider notebooks receiving at least five community votes. We additionally require that the associated data remain accessible through the Kaggle data interface so that the analytical workflow can be reconstructed in an isolated environment.

Data Availability and Resource Constraints. For each notebook, we inspect the files exposed by its mounted datasets or competitions. We exclude a candidate if any required data file is at least 1 GiB, which keeps subsequent sandbox execution practical while avoiding a restriction on the total size of multi-file datasets. Model weights, package archives, and other non-data artifacts are not treated as analytical data and are not downloaded as part of task construction.

Deduplication and Diversity Control. We apply deterministic deduplication before downstream profiling. Duplicate notebook identities are collapsed, and notebooks with identical content fingerprints are retained only once. To prevent a small number of popular datasets or domains from dominating the candidate pool, we retain at most five notebooks per dataset after the initial hard filtering. After acquisition filtering, we retain **10,701 notebooks** for preparation.

B.2 PREPARATION

Workflow Sanitization. Real-world notebooks frequently interleave descriptive analysis with predictive modeling, exploratory trials, visualization, and implementation-specific bookkeeping. Since ITERACT-BENCH focuses on data processing and analytical reasoning rather than model training, we sanitize each notebook before task construction.

Specifically, we identify model-training operations and computations that depend on them, and exclude these operations from the analytical source used by subsequent construction agents. We preserve data loading, cleaning, joins, derived columns, filtering, aggregation, statistical analysis, and other preparation steps required by downstream analytical computations. Inspection and visualization cells may remain as contextual evidence, but visualization-only workflows are subsequently filtered out. Importantly, sanitization defines an exclusion boundary: later construction stages are not allowed to reconstruct a task from discarded training computations.

Programmatic Screening. We first apply deterministic profiling and sandbox execution to remove candidates that lack sufficient analytical structure or cannot be executed reliably. Table 5 summarizes the main screening criteria used in our final configuration.

Candidates are executed against locally mounted copies of their data. Missing inputs, unresolved package dependencies, empty required artifacts, or other execution failures prevent a workflow from being treated as a validated candidate. For executable workflows, we additionally check that the retained analysis forms a non-trivial dependency structure rather than a collection of isolated plotting or inspection operations. These programmatic filters reduce the pool from **10,701** to **7,502 workflows**.

LLM-Based Semantic Screening. Properties such as whether an analysis contains meaningful stateful dependencies or can be converted into a hard-but-fair agent task are difficult to determine through static rules alone. We therefore complement the deterministic filters with an LLM-based semantic rubric. The screening model is DeepSeek-V4-Pro (DeepSeek-AI, 2026) with temperature 0, and receives the sanitized notebook rather than the original training-containing workflow.

Table 5: Main programmatic screening criteria used during workflow preparation.

Criterion	Requirement
Code cells	≥ 3
Executable code cells	≥ 2
Meaningful analytical cells	≥ 3
Longest analytical dependency path	≥ 2
Distinct analytical operation types	≥ 2
Training-related computation ratio	≤ 0.40
Visualization-only ratio	≤ 0.60
External network / secret dependency	Not allowed
Tabular data access	Required
Sandbox execution timeout	600 seconds

Table 6: Semantic workflow-screening rubric.

Dimension	Weight	Criterion
Constructability	4	The retained workflow can be reconstructed locally and contains a closed analytical chain.
Stateful Interaction	5	Later analysis meaningfully consumes states or artifacts produced by earlier analysis.
Hard-but-Fair Reasoning	5	The workflow contains non-trivial but recoverable analytical choices, such as populations, alignment rules, or entity keys.
Exact Verifiability	4	The analytical outcome can be represented by a unique structured answer rather than open-ended prose or visualization.
Shortcut Resistance	2	Solving the task requires actual computation rather than simply copying an answer already exposed in the notebook.

The model scores each workflow from 0 to 5 along five dimensions: Constructability (C), Stateful Interaction (S), Hard-but-Fair Reasoning (H), Exact Verifiability (E), and Shortcut Resistance (R). We compute the weighted score

$$S_{\text{prep}} = 4C + 5S + 5H + 4E + 2R,$$

whose maximum value is 100. Table 6 summarizes the intended meaning of each dimension.

A workflow enters the construction pool only if $S_{\text{prep}} \geq 70$, $C \geq 3$, $E \geq 3$, and no blocking issue is identified. Blocking issues include, for example, an analytically incomplete workflow, outputs that cannot be made objectively verifiable, dependence on removed model-training results, unavailable execution resources, an ungrounded analytical recipe, or unresolved ambiguity. Candidates scoring between 50 and 69 are treated as borderline cases and are not passed to full task construction. This hybrid procedure yields **2,209 validated analytical workflows**.

The screening score is used only for selecting source workflows. It is conceptually and operationally separate from the exact verifier used to evaluate benchmark agents.

B.3 Multi-Agent Task Construction

From the validated workflow pool, 2,209 workflows enter full multi-agent task construction.

We formulate construction as an iterative Builder–Reviewer process with two explicitly separated phases. All Builder and Reviewer agents use DeepSeek-V4-Pro (DeepSeek-AI, 2026) within the Pi agent framework, with reasoning effort set to `xhigh` and temperature 0.6. Programmatic validators are inserted between agentic steps wherever the desired property can be checked deterministically. A failed validation or review triggers a scoped repair followed by re-validation rather than an unrestricted regeneration of the task.

B.3.1 PHASE I: COMPUTATIONAL TRUTH

The first phase determines the analytical computation that the task is intended to evaluate. Its purpose is to establish executable and semantically grounded *computational truth* before any interactive disclosure behavior is designed.

Analytical Stage Construction. A notebook cell is not treated as an analytical stage by default. Notebook cells provide grounded evidence about analytical intent and data flow, but their implementation granularity is often inappropriate for evaluation. The Builder therefore first identifies atomic analytical operations and then converts them into meaningful user-facing stages.

We apply two complementary principles. First, *semantic collapse* merges an intermediate operation into its downstream consumer when the intermediate result is immediately consumed, has no independent analytical value, is not a branch or state-reuse checkpoint, and is not needed as a diagnostic witness. For example, temporary renaming, sorting, inspection, or intermediate DataFrame operations are not promoted to stages solely because they appear in separate notebook cells.

Second, *verification-preserving splitting* prevents excessive merging when a final answer alone cannot distinguish important classes of analytical errors. In such cases, the Builder may separate an intermediate analytical checkpoint or expose an additional structured witness. This preserves the ability of later verification and error analysis to distinguish, for example, an incorrect population from an incorrect metric computation.

Visualization-only operations, inspection-only branches, repeated exploratory trials, discarded training computations, and analytical branches whose intent cannot be recovered reliably are excluded rather than converted into artificial stages.

Grounding and Minimal Repair. We treat the source notebook as evidence of the intended analysis, not as an infallible implementation. Real notebooks occasionally contain evident implementation mistakes, such as a mismatched join key, entity-index misalignment, or a local typographical error. The Builder may repair such an operation only when the correction is minimal and directly supported by the notebook context or underlying data while preserving the original analytical intent.

In contrast, the Builder is not allowed to invent new metrics, thresholds, datasets, or analytical conventions merely to make a workflow constructable. When multiple materially different repairs are equally plausible and the intended computation cannot be uniquely grounded, the affected branch is discarded. This prevents task construction from silently replacing a real analytical workflow with a newly synthesized one.

Reference Execution and Gold Generation. For each retained stage, the Builder constructs an executable reference implementation together with its dependencies and structured output schema. The resulting stage graph explicitly records which downstream stages consume artifacts or analytical states produced upstream.

Reference implementations are executed against the benchmark data in an isolated environment to materialize intermediate states and gold outputs. Programmatic validation checks execution success, answer-schema consistency, dependency-DAG consistency, grounding to the selected source workflow, and cross-stage state compatibility. Where deterministic behavior is expected, the reference workflow is re-executed from a clean state to verify reproducibility.

Independent Computational Review. After programmatic validation, an independent Reviewer Agent is started in a new session with a separate prompt and does not inherit the Builder's dialogue. The Reviewer has access to the sanitized notebook, benchmark data, constructed stage specification, reference implementation, executed outputs, dependency information, and programmatic validation report. It does not modify the task directly.

The Reviewer independently re-executes the reference computation from a clean state and checks whether the implementation actually satisfies the intended analytical obligation, rather than merely reproducing a stored numerical answer. The review additionally examines population definitions, units, filtering and aggregation semantics, ordering conventions, entity alignment, and cross-stage

state dependencies. Detected problems are returned to the Builder as structured findings, after which the affected component is revised and the deterministic validation is rerun.

Freezing Computational Truth. A task proceeds to the interaction-design phase only after its stage boundaries, dependency structure, reference computation, gold outputs, and grounding information pass both programmatic validation and independent semantic review. These artifacts are then frozen as the task's *computational truth*.

This freeze establishes a strict boundary between the two construction phases. Phase II may change how the task is communicated to an agent, but it may not alter stage boundaries, the analytical computation, cross-stage dependencies, or gold outputs in order to make the interaction contract easier to construct.

B.3.2 Phase II: Interaction Contract

The second phase determines how the frozen analytical task is exposed through interaction. Its purpose is therefore different from Phase I: rather than asking whether the underlying computation is correct, Phase II asks whether an agent can recover the information needed to perform that computation through a realistic interaction protocol without being given a complete analytical recipe upfront.

Initial Request and Full Task Specification. For each analytical stage, the Builder constructs two views of the task. The *Initial Request* contains the user-facing analytical objective, essential context, and required output form. The *Full Task Specification* contains the complete set of answer-affecting requirements needed to determine the intended result.

Requirements omitted from the Initial Request are not arbitrary hidden facts. They are analytical conventions that a real user may leave underspecified initially but that must be resolved before the answer is uniquely determined. Typical examples include population definitions, formula coefficients, thresholds, candidate pools, ranking conventions, tie-breaking rules, rounding policies, and similar choices that can change the gold output. Such requirements remain part of the same stage specification and may be disclosed through clarification or correction by the user simulator.

Conversely, the Initial Request must not be made artificially vague. The core business objective, necessary context, and expected answer structure remain visible from the beginning. This yields two symmetric failure modes that the construction procedure explicitly checks: an overly complete Initial Request reduces the benchmark to executing a fully specified recipe, whereas an under-specified interaction contract may leave answer-affecting choices unrecoverable.

Programmatic Contract Validation. Before semantic review, deterministic validators check the interaction representation for structural and cross-artifact consistency. These checks verify, among other properties, that answer fields are covered by the specification, disclosed requirements do not contradict the frozen computational truth, cross-stage dependencies reference valid upstream states, and the Initial Request does not directly expose reference outputs or otherwise leak the gold answer.

Blind Contract Review. We then perform an independent, surface-first contract review using a fresh Reviewer Agent. The review deliberately separates the information available before and after user interaction.

First, the Reviewer sees only the Initial Request, answer schema, dependency state, and the relevant benchmark data or data schema. It records which answer-affecting decisions remain unresolved and what information a reasonable solver would need to request. The existence of missing information at this point is not itself a failure—it is an intended property of interactive stages.

The Reviewer is then given the complete specification according to the intended disclosure rules and independently implements the analytical stage in dependency order. During this independent implementation, it does not have access to the reference code or gold output. Only after producing its own result is the implementation compared against the frozen reference computation.

A contract passes this review when the combined information is sufficient to recover a unique intended analysis, remains implementation-neutral, exposes all answer-affecting requirements when needed,

does not reveal the solution directly, and allows the independent implementation to align with the frozen computation. The Reviewer also checks that the Initial Request has not become so complete that clarification is unnecessary by construction.

This procedure is a *blind contract probe*, rather than a full benchmark evaluation trajectory. Its purpose is to validate the information contract itself; live agent–user interaction is evaluated subsequently in the benchmark runtime.

Scoped Revision and Compilation. When Phase-II validation detects a missing, contradictory, or overly revealing requirement, the Builder may revise only the interaction contract. The frozen analytical computation and gold outputs remain unchanged. All affected programmatic checks and the blind contract review are then rerun. We allow up to three scoped repair rounds before failing closed on a candidate that still contains blocking issues.

Tasks that pass both construction phases are compiled into runnable Harbor tasks in sandboxed environments. After the two construction gates, 310 tasks enter post-construction quality assurance.

B.4 Quality Assurance

Construction-time quality control relies primarily on deterministic invariants, executable validation, and the two independent LLM review gates described above. We do not assume that every generated task must additionally receive a full manual review. Instead, human inspection is targeted using failures observed during post-construction execution.

Batch Evaluation. We execute the constructed tasks across the model–harness configurations used in our benchmark experiments. This serves two purposes: it exercises tasks under heterogeneous agent behaviors and runtime implementations, and it surfaces failures that may indicate a defect in the task rather than an inability of a particular agent.

A low score alone never causes a task to be modified or removed. Only failures for which subsequent attribution identifies evidence of a benchmark-side defect are eligible for task repair.

Fault Attribution. Every failed exact-match trajectory considered during QA is processed by a separate fault-attribution pipeline. The pipeline first applies deterministic system auditing before any LLM-based diagnosis. This audit identifies directly observable execution failures, including provider rate limits or quota exhaustion, runtime exceptions, message truncation, delivery failures, and incomplete protocol execution. Confirmed infrastructure failures are attributed to the execution system rather than reassigned to the evaluated agent.

For failures requiring semantic diagnosis, we construct a restricted evidence package rather than exposing the complete raw trial. The package contains the relevant interaction contract, user–agent dialogue, submitted answer, localized verifier discrepancies, and the result of the deterministic system audit. The attribution model is not given the source notebook, reference implementation, raw benchmark dataset, or complete gold answer. Only the evidence necessary to diagnose the observed discrepancy is exposed.

We use Qwen3.8-Max in an isolated Pi environment for fault attribution. A task-level auditor first examines the authored task contract, while stage-level investigators inspect individual failed interactions. A separate adjudicator then consolidates the evidence into a structured trial-level report. Each trial receives exactly one *primary* attribution, whose owner is one of *agent*, *execution system*, or *task*. When the responsible owner can be identified but the evidence is insufficient to select a finer-grained subtype, the attribution is retained as an owner-specific inconclusive case rather than introducing a fourth top-level owner.

Table 7 summarizes the fine-grained taxonomy. Agent-owned failures distinguish failures to acquire missing requirements from failures to follow available requirements, select the correct data, execute the intended computation, or satisfy the submission protocol. Execution-system failures cover infrastructure faults as well as user-simulator errors: the latter are assigned only when the authored contract uniquely determines the gold-consistent rule but the simulator provides incorrect or misleading guidance. Task-owned failures instead indicate defects in the authored benchmark

contract itself, such as an answer-affecting requirement that cannot be recovered from the available specification or a conflict between disclosure and frozen gold.

First-divergence adjudication. Primary attribution follows a fixed first-divergence order: *authored contract → user-simulator disclosure → model-seen context → agent action → computation → submission → verification*. The earliest layer at which the trajectory becomes inconsistent with the validated task contract determines the primary attribution. This precedence prevents later symptoms from replacing an earlier root cause; for example, a wrong downstream computation caused by an unresolved upstream requirement remains rooted in Requirement Acquisition rather than being relabeled as an independent computation failure.

Propagation. We separately record downstream failures attributed to earlier faults. Such failures are marked as non-independent, and `propagated_from_turn` identifies the attributed upstream fault turn. The subset directly linked to the trial's primary root is analyzed separately in Appendix D.4.3. Accordingly, `AGENT_STATE_MANAGEMENT` is not used as a primary fault category. When incorrect analytical state is reused downstream, the original substantive error retains the primary code and subsequent failures are recorded as propagated consequences. This avoids repeatedly counting a single upstream mistake as multiple independent state-management failures.

Task-side remediation. Fault attribution is diagnostic rather than an automatic task-editing mechanism. Any task-owned attribution is manually reviewed during benchmark QA. Confirmed issues are repaired when the intended task can be recovered unambiguously; otherwise, the task is removed. All experimental results reported in the paper are computed only on the final validated benchmark after this QA process. Intermediate QA-stage failures therefore do not enter the reported benchmark results.

Targeted Human Review. All cases attributed to the task are manually inspected by one researcher. The reviewer compares the Initial Request, Full Task Specification, interaction trace, verifier discrepancy, and frozen computational truth to determine whether the observed failure is caused by a genuine benchmark defect.

If the specification is valid and uniquely recoverable, the task is retained even when multiple evaluated agents fail. If the interaction contract omits or contradicts a requirement that is already supported by the frozen computation, the issue is treated as a recoverable specification defect. Cases for which the intended analytical rule cannot be established uniquely are conservatively retired rather than resolved through an arbitrary author choice.

Repair and Retirement Policy. Recoverable defects are repaired at the narrowest possible scope. For example, a missing answer-affecting convention may be added to the Full Task Specification, or an Initial Request that contradicts the frozen computation may be corrected. Such repairs do not modify the frozen reference computation or gold output and must pass the interaction-contract validation again.

If repairing a case would require changing the intended computation, inventing a new formula or convention, or selecting arbitrarily among multiple plausible interpretations, the task is retired from evaluation. Execution-system failures instead trigger fixes to the execution infrastructure without changing the task, while agent-attributed failures do not trigger task modification.

Post-construction QA begins with 310 candidate tasks. Batch execution and fault attribution flag 120 distinct tasks for potential task-side defects. Each flagged task is manually inspected to determine whether the issue is recoverable. Recoverable tasks are manually repaired and subjected to renewed execution, validation, and fault attribution; tasks with irrecoverable defects are excluded. For retained tasks, this process continues until no unresolved task-side defect remains in the inspected QA runs.

We then apply a difficulty filter: a task is excluded if all Qwen3.7-Max, Qwen3.8-Max, GLM-5.2, and DeepSeek-V4-Flash evaluated with Pi-Agent, successfully complete it in all three valid screening trials per model. Every remaining candidate undergoes manual review for analytical plausibility, specification consistency, meaningful stage dependencies, and reference-answer validity. Tasks with unresolved defects in any of these aspects are excluded.

Table 7: Fault-attribution taxonomy. Each failed trial receives exactly one primary owner: *execution system*, *task*, or *agent*. Agent-owned primary faults use the same five categories reported in Figure 13. Downstream propagation is recorded separately and does not replace the primary root cause.

Owner	Subcategory	Definition
Execution system	Provider / Quota Failure	Provider rate limit, quota exhaustion, or related service failure terminates the trial.
	Harness Truncation	The harness truncates a model or user message.
	Delivery Gap	Authored or simulated-user content is produced but does not reach the model-visible interaction history.
	Runtime Exception	A runtime or Harbor exception terminates or corrupts execution.
	Incomplete Protocol	The interaction terminates with an unfinished turn or without a clean protocol close.
	User-Simulator Error	The authored contract uniquely determines the gold-consistent rule, but the user simulator provides incorrect or misleading guidance.
	Other	Other execution-system failure not covered above.
Task	Missing / Undisclosed Requirement	An answer-affecting requirement is neither uniquely specified nor naturally recoverable through clarification.
	Disclosure–Gold Conflict	The surfaced specification or interaction contract conflicts with the frozen gold result.
	Unsupported Gold-Specific Mapping	The authored task requires a gold-specific mapping that is not uniquely determined by the available contract.
	Other	Other defect in the authored benchmark contract.
Agent	Requirement Acquisition	A gold-consistent requirement is available or recoverable through clarification, but the agent fails to acquire it.
	Requirement Adherence	An available or disclosed analytical requirement is not followed.
	Tool / Data Selection	The agent selects an incorrect population, table, column, join, filter, or related data operation.
	Computation Error	The intended requirement and data are identified correctly, but the formula, algorithm, or arithmetic computation is incorrect.
	Submission Protocol	The agent violates the required answer envelope, field names, or submission structure and fails to recover.

The sequential exclusion stages remove 6 tasks for irrecoverable defects, 163 tasks under the difficulty filter, and 55 tasks during final manual review. Each excluded task is counted once, at the stage where it is removed. These exclusions total 224 tasks, leaving 86 tasks in the final release.

For the retained benchmark, we repeat the relevant validation and QA procedures until no unresolved task-attributed failure remains among the evaluated model–harness configurations. The final release contains **86 tasks** comprising **554 analytical stages**.

C Experimental Details

C.1 Model and Harness Configuration

We evaluate seven frontier LLMs under Pi-Agent. Four of these models are additionally evaluated under OpenClaw, Codex, and Claude Code, yielding 19 model–harness configurations in total.

Each harness is based on its official implementation and interacts with ITERACT-BENCH through the unified MCP-based interaction interface. The benchmark runtime provides access to the required task files and data, while preserving the default analysis tools provided by each agent framework. Detailed harness implementations and versions are summarized in Table 8.

Table 8: Agent framework implementations used in evaluation.

Harness	Implementation	Version
Claude Code	`@anthropic-ai/claude-code`	2.1.220
Codex	`@openai/codex`	0.146.0
OpenClaw	`openclaw`	2026.7.1-2
Pi-Agent	`@earendil-works/pi-coding-agent`	0.84.1

C.2 Inference and Runtime Configuration

All models use their default API sampling configuration with temperature set to 1.0 and reasoning effort set to high. The user simulator is implemented with GLM-5.2. Unless otherwise specified, all other model and harness configurations follow their default API and runtime settings.

The benchmark runtime enforces both episode-level and interaction-level limits. The default configuration uses a maximum of 500 runtime steps per trial, a maximum of 30 execution errors, and a maximum of 10 agent–user exchanges per analytical stage.

C.3 Evaluation Protocol

Each model–harness configuration is evaluated until three valid trials are obtained. Trials affected by API failures are considered invalid and excluded from evaluation. Such trials are rerun to obtain the required number of valid trials.

All reported results are based on valid trials without API failures, runtime failures, harness launch failures, or infrastructure errors. Failures caused by the agent itself, including failing to produce a valid submission, exceeding interaction budgets, or timing out during execution, are treated as unsuccessful trials rather than excluded cases.

The primary metric is Average Pass Rate (APR). The metric definition and aggregation procedure are provided in Appendix A.4. Specifically, the task-level pass rate is computed over valid trials, and the final benchmark score is averaged across tasks according to Eq. 5 and Eq. 4.

Confidence Intervals. We use a task-level cluster bootstrap to estimate uncertainty in the reported accuracy statistics and accuracy differences. For each analysis, we draw 10,000 bootstrap samples using random seed 42. Each sample draws tasks with replacement from the relevant task set, retaining the original number of tasks. Whenever a task is selected, all of its associated stages, evaluated configurations, and valid trials are retained together; a task selected multiple times contributes with the corresponding multiplicity. We recompute the statistic in each bootstrap sample and use the 2.5th and 97.5th percentiles as the endpoints of the 95% confidence interval. Reported point estimates are computed directly from the original observations.

The statistic and relevant task set differ across analyses:

- **Model-level APR (Figure 4).** For each model–harness configuration, we resample the 86 benchmark tasks. Each task's pass rate is the mean success indicator over its three valid trials, and APR is the mean of these task-level pass rates.
- **Task-level workflow complexity (Figure 5b, top, and Figure 11).** Tasks are grouped by maximum DAG depth or workflow length, respectively, and resampled separately within each group. For each task, we first average success indicators over valid trials within each configuration and then equally average across the 19 configurations in the full cross-harness comparison. The group statistic is the mean of these task-level scores.
- **Stage-level workflow complexity (Figure 5b, bottom).** For each stage-depth group, we resample tasks containing at least one stage in that group. Each task contributes its counts of passed and evaluated stage–trial instances, pooled across the 19 configurations. The statistic is the ratio of the total passed count to the total evaluated count in the bootstrap sample. Unreached stages are included as failures. For the 6+ group, counts at depths of at least six are combined within each task before resampling.

Table 9: APR (%) under different specification and interaction settings with Pi-Agent.

Setting	Qwen3.7-Max	Qwen3.8-Max
No Interaction	1.2	4.3
Clarification Only	2.7	36.8
Correction Only	29.5	60.5
Clarification + Correction	**33.3**	**70.5**
Full Specification	80.2	85.7

- **Matched repeated-trial analysis (Table 12 and Figure 12).** We retain stage–configuration groups whose trials differ in whether the agent asks before its first submission. Within each group, we compute the difference in mean outcome between trials with and without this behavior. The reported statistic is the unweighted mean of these group-level differences. We resample tasks containing eligible groups, retaining all eligible groups and their trials within each sampled task. The same procedure is applied to the overall analysis and each model or harness subgroup.

All intervals are pointwise rather than simultaneous confidence bands. They quantify task-sampling uncertainty conditional on the observed trials and evaluated configurations. Error bars and shaded regions represent these intervals.

D Additional Experimental Results

D.1 Ablation of Interaction Channels

To examine the contributions of clarification and correction, we evaluate Qwen3.7-Max and Qwen3.8-Max under Pi-Agent with four interaction settings. All four settings begin with the same Initial Request. *No Interaction* provides no subsequent clarification or corrective feedback. *Clarification Only* allows the user simulator to answer agent questions but disables correction. *Correction Only* allows corrective feedback on submissions but disables clarification. *Clarification + Correction* enables both channels and corresponds to the standard On-Demand Specification setting. We also report Full Specification results for reference.

Table 9 reveals substantial differences in how models benefit from clarification. Compared with No Interaction, Clarification Only improves Qwen3.7-Max by just 1.6 percentage points, whereas Qwen3.8-Max gains 32.6 points. Thus, providing an opportunity to ask questions does not ensure that an agent effectively obtains and uses the missing requirements.

Correction Only achieves higher APR than Clarification Only for both models, showing the substantial contribution of feedback after submission. Nevertheless, enabling clarification alongside correction provides further gains of 3.9 points for Qwen3.7-Max and 10.1 points for Qwen3.8-Max. Both models therefore benefit from access to both channels, although their relative gains differ. Even with both channels available, performance remains below Full Specification, reinforcing the challenge of obtaining and applying analytical requirements through interaction.

D.2 Additional Workflow Complexity Analysis

We complement the dependency-depth analysis in Section 3.3.2 by examining performance as a function of workflow length. Figure 11 reports task-level APR grouped by the number of analytical stages, aggregating results across all evaluated model–harness configurations.

APR generally decreases as workflows become longer, falling from approximately 54% for tasks with 2–3 stages to 23% for those with 15 or more stages. The trend is not strictly monotonic, with modest variation among the intermediate groups. Together with the dependency-depth results, this pattern suggests that maintaining analytical correctness becomes more challenging over extended workflows. Workflow length and dependency depth capture different structural properties, although this grouped analysis does not isolate their individual effects.

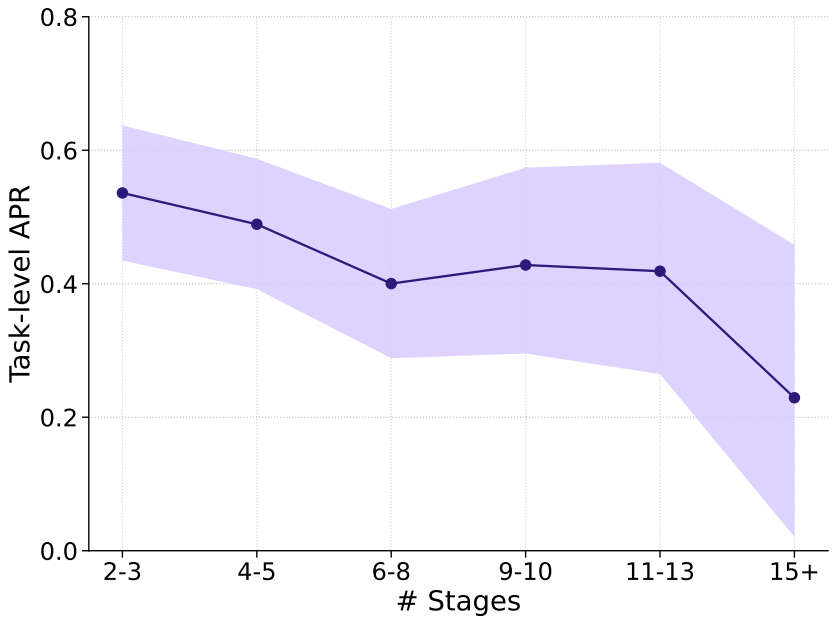


Figure 11: Task-level APR by workflow length, measured by the number of analytical stages. Results are aggregated across all evaluated model–harness configurations. Performance generally declines as workflows become longer.

Table 10: Complete interaction statistics across model–harness configurations. All accuracy and rate columns are percentages. Ask, Corr., Exch., and Sub. are averages per analytical stage.

Model	Harness	APR	Stage Acc.	First Acc.	Ask-Before	Clarify	Correct	Proactive	Ask	Corr.	Exch.	Sub.	Budget
Qwen3.7-Max	Pi-Agent	33.3	74.8	56.6	7.5	7.6	69.7	8.0	0.09	0.93	2.01	1.92	0.9
Qwen3.7-Max	OpenClaw	29.8	67.3	57.1	5.8	6.2	64.0	7.1	0.07	0.84	1.81	1.74	0.6
Qwen3.7-Max	Codex	27.9	69.0	54.4	7.1	7.6	73.6	8.2	0.09	0.99	2.09	2.00	1.1
Qwen3.7-Max	Claude Code	27.1	67.5	50.9	6.8	6.9	64.2	7.8	0.08	0.87	1.90	1.82	0.5
Qwen3.8-Max	Pi-Agent	70.5	88.4	81.8	55.7	54.7	38.0	57.6	0.61	0.44	2.02	1.40	0.9
Qwen3.8-Max	OpenClaw	50.8	85.2	70.0	40.3	40.5	51.1	40.4	0.44	0.64	2.06	1.62	0.9
Qwen3.8-Max	Codex	68.2	85.0	79.1	56.6	55.3	37.5	58.3	0.65	0.44	2.03	1.38	0.9
Qwen3.8-Max	Claude Code	58.5	86.4	71.4	44.6	44.4	50.1	44.8	0.49	0.59	2.08	1.59	1.0
GLM-5.2	Pi-Agent	53.9	81.7	72.3	32.8	31.9	60.1	33.2	0.37	0.71	2.01	1.65	0.5
GLM-5.2	OpenClaw	26.4	70.1	53.5	5.3	5.4	71.8	5.8	0.06	0.95	2.00	1.94	1.0
GLM-5.2	Codex	40.7	77.1	60.1	11.8	12.8	71.5	13.9	0.16	0.95	2.13	1.97	1.1
GLM-5.2	Claude Code	57.0	86.2	70.9	30.7	30.3	64.7	30.4	0.34	0.76	2.11	1.77	0.9
DeepSeek-V4-Flash	Pi-Agent	46.9	73.3	66.4	22.2	22.2	60.6	25.5	0.26	0.74	1.89	1.63	1.0
DeepSeek-V4-Flash	OpenClaw	35.3	73.1	62.7	16.2	16.3	66.7	18.1	0.18	0.82	1.92	1.74	0.6
DeepSeek-V4-Flash	Codex	42.6	80.7	66.1	30.7	30.2	69.0	29.0	0.36	0.85	2.22	1.86	1.0
DeepSeek-V4-Flash	Claude Code	41.9	79.7	56.7	22.2	22.4	63.9	22.6	0.25	0.85	2.10	1.84	1.0

D.3 Additional Interaction Behavior Results

Complete interaction statistics. Table 10 reports the complete interaction statistics for all model–harness configurations. In addition to task-level APR and final-stage accuracy, we report first-submission accuracy and several measures of interaction behavior. *Ask-Before* denotes the fraction of stages in which the agent asks at least one question before its first submission. *Clarify* and *Correct* denote the fractions of stages receiving at least one clarification and correction, respectively. The *Proactive Ratio* is computed over interaction events as the number of clarification events divided by the total number of clarification and correction events. Ask, Corr., Exch., and Sub. report the average number of questions, corrections, interaction exchanges, and submissions per stage, respectively.

System-level interaction–performance associations. Table 11 reports Spearman correlations between configuration-level interaction statistics and APR. The results reinforce the distinction between interaction strategy and interaction volume: measures of proactive specification acquisition are strongly associated with APR, whereas the total number of exchanges is not.

Table 11: Spearman correlations between interaction behavior and APR across the 19 model–harness configurations.

Interaction Metric	ρ with APR	p
Ask-before-first-submit rate	0.963	2.2×10^{-9}
Clarification rate	0.962	2.5×10^{-9}
Proactive interaction ratio	0.965	1.7×10^{-9}
Correction rate	-0.785	3.1×10^{-4}
Avg. questions per stage	0.957	6.9×10^{-9}
Avg. corrections per stage	-0.875	8.9×10^{-6}
Avg. exchanges per stage	0.328	0.215

Table 12: Matched repeated-trial analysis of proactive clarification. Each comparison holds the task stage and model–harness configuration fixed. ΔFinal denotes the percentage-point difference in accuracy of finalized stage answers between trials that ask before their first submission and those that do not. ΔCorrect denotes the percentage-point difference in the probability of receiving a subsequent correction. ΔExch. denotes the difference in mean interaction exchanges per stage. Brackets show 95% task-level bootstrap confidence intervals.

Group	Matched Groups	ΔFinal Acc.	ΔCorrect	ΔExch.
Overall	1,857	+12.3 [+9.6, +15.2]	−27.9 [−32.7, −22.9]	+0.59 [+0.52, +0.65]
By model				
Qwen3.7-Max	206	+30.8 [+18.4, +42.7]	−11.2 [−19.3, −3.7]	+0.79 [+0.52, +1.05]
Qwen3.8-Max	777	+9.3 [+6.2, +12.6]	−35.6 [−41.8, −29.2]	+0.53 [+0.42, +0.64]
GLM-5.2	466	+6.8 [+2.9, +11.0]	−35.4 [−43.4, −27.1]	+0.42 [+0.33, +0.52]
DeepSeek-V4-Flash	408	+15.0 [+10.0, +20.3]	−13.2 [−22.3, −4.0]	+0.77 [+0.62, +0.94]
By harness				
Pi-Agent	540	+13.9 [+9.6, +18.7]	−28.5 [−35.4, −21.5]	+0.68 [+0.56, +0.79]
OpenClaw	371	+17.1 [+10.3, +23.4]	−24.0 [−31.5, −15.5]	+0.66 [+0.54, +0.80]
Codex	404	+9.7 [+4.9, +14.5]	−28.6 [−35.3, −21.8]	+0.42 [+0.31, +0.52]
Claude Code	542	+9.4 [+5.1, +13.5]	−29.6 [−35.5, −23.2]	+0.57 [+0.46, +0.68]

Matched repeated-trial analysis. The configuration-level correlations may be confounded by differences in underlying model and harness capability. We therefore additionally compare repeated trials of the same analytical stage under the same model–harness configuration. Specifically, we retain a (task, stage, model, harness) group only when its repeated trials differ in whether the agent asks before its first submission. This yields 1,857 matched groups spanning all 86 tasks. For each group, we compute the difference between trials with and without ask-before-first-submit behavior. Confidence intervals are obtained by task-level bootstrap resampling (10,000 samples).

Table 12 provides the overall results and breakdowns by model and harness. The positive association with both first-submission and final-stage correctness is observed across every model and every harness. Proactive trials also receive fewer subsequent corrections, although they consume slightly more total interaction exchanges.

Matched Within-Stage Analysis. The configuration-level correlation in Figure 6(b) does not by itself isolate the association between proactive interaction and analytical correctness: different models and harnesses may both interact more proactively and achieve higher APR for other reasons. We therefore exploit stochastic variation across repeated trials and compare runs of the same analytical stage under the same model–harness configuration.

Among 1,857 stage–configuration groups in which ask-before-first-submit behavior varies across trials, proactively asking before the first submission is associated with a 20.3 percentage-point increase in first-submission accuracy and a 12.3-point increase in final-stage accuracy (Figure 12). The corresponding 95% task-level bootstrap confidence intervals are $[16.7, 23.7]$ and $[9.6, 15.2]$, respectively. Proactive trials also receive fewer subsequent corrections, suggesting that clarification before execution is associated with fewer specification violations exposed after submission. Breakdowns by

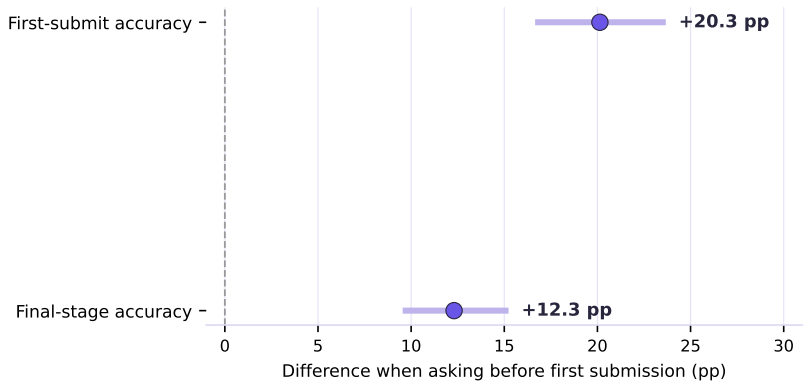


Figure 12: **Matched within-stage association between proactive clarification and correctness.** Within the same analytical stage and model–harness configuration, trials that ask before their first submission achieve higher first-submission and final-stage accuracy. Error bars denote 95% task-level bootstrap confidence intervals.

model and harness are reported in Table 12. While this observational comparison does not establish causality, the association persists after holding the analytical stage, model, and harness fixed.

D.4 Failure Analysis

We perform trial-level fault attribution on failed runs using the procedure described in Appendix B.4. For agent-owned failures, we consider five primary categories: *Requirement Acquisition*, *Requirement Adherence*, *Computation Error*, *Submission Protocol*, and *Tool/Data Selection*. We first examine how the distribution of these failures changes across models and harnesses, and then analyze how primary failures propagate through dependent analytical stages.

D.4.1 Primary Fault Composition

Figure 13 presents two controlled comparisons. Holding the harness fixed to Pi-Agent, the most pronounced difference between Qwen3.7-Max and Qwen3.8-Max lies in Requirement Acquisition: it accounts for 70.2% of Qwen3.7-Max's agent-owned primary faults, but only 41.1% of Qwen3.8-Max's. The residual failures of Qwen3.8-Max are correspondingly more distributed across Requirement Adherence, Computation Error, and Submission Protocol. This shift is consistent with the interaction analysis in Section 3.3.3, where Qwen3.8-Max asks for missing information substantially more proactively than Qwen3.7-Max. Qwen3.8 is also reported to improve agentic planning and the handling of environment feedback (Qwen Team, 2026b); in our setting, these capabilities appear to manifest as more proactive specification acquisition.

Holding the model fixed to GLM-5.2 reveals a similar sensitivity to the agent harness. Requirement Acquisition accounts for 65.8% of agent-owned primary faults with OpenClaw, compared with 50.0% with Pi-Agent. This difference again aligns with interaction behavior: GLM-5.2 asks before its first submission in 32.8% of stages with Pi-Agent, but only 5.3% with OpenClaw. Thus, the ability to acquire unresolved analytical requirements is shaped not only by the underlying model but also by the interaction policy imposed by the harness.

D.4.2 Harness Prompt Analysis

To better understand the harness-dependent difference observed for GLM-5.2, we inspect the system-level instructions added by Pi-Agent and OpenClaw. The benchmark task prompt and interaction protocol are unchanged across harnesses; the relevant difference arises from additional harness instructions supplied to the model. The quotations below are taken from the compiled OpenClaw system prompt logged in GLM-5.2 production trials (OpenClaw 2026.7.1-2).

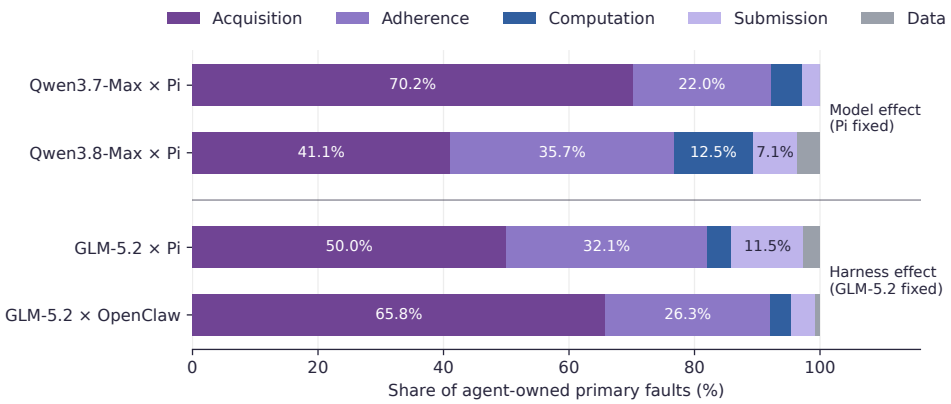


Figure 13: **Primary fault composition under controlled model and harness comparisons.** Holding Pi-Agent fixed, Requirement Acquisition accounts for a substantially smaller share of primary faults for Qwen3.8-Max than for Qwen3.7-Max. Holding GLM-5.2 fixed, Requirement Acquisition faults are more prevalent under OpenClaw than under Pi-Agent.

OpenClaw introduces two layers that favor acting before asking. The first is a fixed *Execution Bias* block in the harness system prompt:

> "Actionable request: act in this turn."
> "Non-final turn: use tools to advance, or ask for the one missing decision that blocks safe progress."

The second is the default workspace persona injected from `SOUL.md`:

> "Be resourceful before asking. Read the file, check the context, search for it. Come back with answers, not questions."

Together, these instructions encourage asking only when missing information blocks further progress. This policy is potentially mismatched with ITERACT-BENCH: many underspecified analytical decisions do not prevent execution because the agent can proceed with a plausible default or implicit assumption, yet these decisions still affect the correct analytical result and should therefore be clarified. A further workspace line, "Don't interrogate," appears only in the first-run `BOOTSTRAP.md` identity script and is not a general restriction on analytical clarification; we therefore do not treat it as part of the same policy.

Pi-Agent, in contrast, appends instructions for following the benchmark interaction protocol, including discovering tools, retrying a timed-out stakeholder call, and not ending the turn while the interaction remains open. It does not introduce a comparable preference against asking questions.

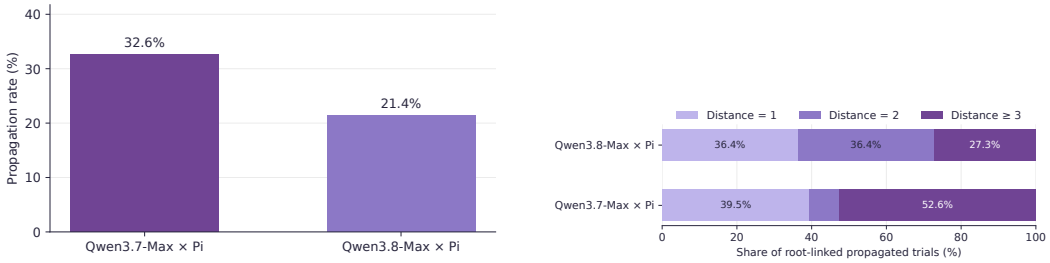
This prompt-level difference provides a plausible mechanism for the pattern in Figure 13: with the same GLM-5.2 model, OpenClaw produces both a substantially lower ask-before-first-submit rate and a higher share of Requirement Acquisition failures. The same model on Claude Code, which also exposes native MCP tools but does not add an act-before-ask bias, exhibits an ask-before-first-submit rate much closer to Pi-Agent. This is consistent with an OpenClaw-specific prompt effect rather than with the mere absence of an explicit ask-first instruction. We treat this prompt inspection as explanatory evidence rather than a causal ablation of the individual instructions.

D.4.3 CROSS-STAGE ERROR PROPAGATION

We next examine how failures propagate across analytical stages in trials with agent-owned primary faults. A trial is considered to exhibit propagation when the fault-attribution report identifies at least one downstream failure as a propagated consequence of an earlier fault rather than an independent error (`independent_failure=false`). We further identify a root-linked subset in which at

Table 13: Cross-stage propagation under Pi-Agent. Root-linked statistics include only propagated failures that can be directly traced to the primary root fault.

	Qwen3.7-Max	Qwen3.8-Max
Agent-owned primary failures	141	56
Trials with propagation	46	12
Propagation rate	32.6%	21.4%
Root-linked propagated trials	38	11
Median propagation distance	3	2
Distance ≥ 3	20 (52.6%)	3 (27.3%)
Acquisition-rooted propagation	27 (71.1%)	2 (18.2%)
Acquisition-rooted, distance ≥ 3	15	1



(a) Propagation rate. **(b) Propagation distance.**

Figure 14: **Cross-stage error propagation under Pi-Agent.** (a) Among trials with agent-owned primary faults, downstream propagation is more frequent for Qwen3.7-Max than for Qwen3.8-Max. (b) Among root-linked propagated failures, Qwen3.7-Max also exhibits longer propagation chains, with a larger fraction reaching distance three or greater.

least one such failure has `propagated_from_turn` pointing directly to the trial's primary root turn. For each root-linked trial, propagation distance is the maximum downstream turn distance from the primary root among these directly linked failures.

Figure 14 summarizes the frequency and extent of propagation under Pi-Agent. Among trials with agent-owned primary faults, downstream propagation is observed in 46 of 141 trials (32.6%) for Qwen3.7-Max, compared with 12 of 56 trials (21.4%) for Qwen3.8-Max. Of these propagated trials, 38 and 11, respectively, contain failures directly linked to the primary root and are included in the propagation-distance analysis. Within this root-linked subset, 20 of 38 Qwen3.7-Max trials (52.6%) reach a distance of at least three, compared with 3 of 11 Qwen3.8-Max trials (27.3%). The corresponding median propagation distances are three and two.

Table 13 further characterizes the role of Requirement Acquisition faults in these root-linked cases. For Qwen3.7-Max, 27 of 38 cases (71.1%) have a Requirement Acquisition primary root, compared with 2 of 11 cases (18.2%) for Qwen3.8-Max. Among these acquisition-rooted cases, 15 for Qwen3.7-Max and 1 for Qwen3.8-Max reach a propagation distance of at least three. These observations show that failures to acquire analytical requirements can have consequences beyond the stage where they first arise: an unresolved requirement can lead to incorrect analytical state that is subsequently reused in dependent computations.

D.5 Effect of the User Simulator

We assess sensitivity to the user simulator by fixing the evaluated agent to Qwen3.8-Max with Pi-Agent and varying the simulator model. Each configuration is evaluated on the same 258 trials. Alongside APR, we report the *USIM-attributed trial rate*: the fraction of all trials whose primary failure is attributed to the simulator by the fault-attribution procedure in Appendix B.4.

Table 14: Simulator sensitivity and human repair, with the evaluated agent fixed to Qwen3.8-Max with Pi-Agent. USIM-attributed trials are original failures primarily attributed to the simulator; percentages use all 258 trials. Recovered trials report full-task successes among these intervened failures. APR after repair substitutes the resumed outcomes into the original evaluation set.

User Simulator	Original APR	USIM-attributed Trials	Recovered Trials	APR after Repair
Qwen3.8-Max	69.8%	24 (9.3%)	8/24	72.9%
Qwen3.7-Max	57.8%	39 (15.1%)	12/39	62.4%
GLM-5.2	70.5%	20 (7.8%)	7/20	73.3%

Sensitivity to simulator choice. Table 14 shows similar original APRs with Qwen3.8-Max and GLM-5.2 as simulators (69.8% and 70.5%), whereas Qwen3.7-Max yields 57.8%. The corresponding numbers of USIM-attributed failures are 24, 20, and 39. These results indicate that simulator choice can affect measured agent performance, motivating a direct examination of the identified simulator errors.

Human repair and continuation. We intervene on all USIM-attributed failed trials for each simulator. Each trial resumes immediately before its first erroneous simulator response, preserving the preceding dialogue and execution state. The response is manually corrected to comply with both the Full Task Specification and the original disclosure policy: only information warranted by the current interaction is provided. Human review continues throughout the remainder of the trial, correcting subsequent simulator errors under the same rules. The agent then completes the remaining stages, and the resulting answers are scored by the original independent verifiers.

We report *APR after repair* by replacing the original outcomes of these intervened trials with their resumed outcomes while retaining all other trial outcomes. This is a supplementary sensitivity analysis; the original APR remains the benchmark result.

Results. Repair recovers full-task success in 8 of 24 flagged trials for Qwen3.8-Max, 12 of 39 for Qwen3.7-Max, and 7 of 20 for GLM-5.2. The resulting APRs are 72.9%, 62.4%, and 73.3%, respectively, corresponding to increases of 3.1, 4.7, and 2.7 percentage points, computed from unrounded counts. Manual review attributes the remaining failures in the repaired continuations to agent errors.

The two stronger simulators therefore yield similar APRs both before and after intervention; after repair, their estimates differ by 0.4 percentage points. For GLM-5.2, used in our main evaluation, the observed change is 7 additional successes out of 258 trials. These results support the stability of the aggregate APR across the two stronger simulators for this agent configuration, while confirming that simulator errors account for some failed trials.