ENTERPRISEWORLD: A REALISTIC WORLD FOR ENTERPRISE LONG-HORIZON AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

In enterprise workflows, agents must carry the results, records, and state changes of earlier operations into later ones, often across systems. Existing benchmarks increasingly capture complex, stateful and long-horizon tasks, but rarely make the dependencies between tasks explicit, verified and controllable, limiting both diagnosis of workflow failures and controlled evaluation of changing task relationships. We therefore introduce **ENTERPRISEFLOW**, a programmatic framework for constructing enterprise workflows around four design principles: *realistic* enterprise operations, *long-horizon* coordination, *structural control* over task dependencies, and *verifiable* local and cross-task outcomes. Using this framework, we construct **ENTERPRISEWORLD**, a benchmark of 200 cross-system workflows with 20 operations each in a shared executable environment. Dependencies are explicitly grounded in results and state changes produced by earlier operations, verified by tracing input origins and testing whether upstream errors affect downstream success, and further supported by a human audit with 97.2% agreement. We also construct paired workflow variants that either modify the required dependency topology or add task/tool distractors while preserving it. Our evaluation measures local outcomes, progress along valid dependency chains, and workflow completion, enabling process-level diagnosis of where execution breaks down. Across five frontier models, GPT-6 Astra completes only 46.5% of full workflows, while completion drops by 32-46 points as execution extends across longer dependency chains. Ablation analysis further shows that supplying the correct intermediate results from earlier operations in the dependency chain helps substantially more than exposing the dependency structure, with most dependency failures originating when those results are produced; matched perturbations also break 14-26% of previously successful workflows.

1 INTRODUCTION

Large language model agents are becoming capable of increasingly complex enterprise operations, from querying business data to interacting with workplace applications (Zhou et al., 2024; Mialon et al., 2023; Xie et al., 2024; Jimenez et al., 2024; Lu et al., 2025a). Progress in agents' ability to carry out such tasks has been supported by methods that interleave reasoning and acting, design interfaces for tool interaction, and introduce executable interventions into the agent loop (Yao et al., 2023; Yang et al., 2024; Ma et al., 2024b; Liu et al., 2026). Real business execution, however, requires this *task intelligence* to translate into *workflow intelligence*: individual operations must remain coordinated through the dependencies that make an entire business process valid. In a payment workflow, for example, a computed amount must inform the appropriate approval, that approval must authorize the corresponding payment, and reconciliation must use the resulting transaction record. Such dependencies may cross systems, span many intervening operations, and require several earlier results to jointly constrain a later decision. Even with the same business objective, a new review requirement or intermediate handoff can change how these operations must be coordinated.

Evaluating this capability requires more than harder individual tasks or longer trajectories. Existing benchmarks support rich tool use, compositional planning, stateful interaction, and long-horizon execution (Yoran et al., 2024; Ma et al., 2024a; Lu et al., 2025b; Wang et al., 2025a; Xu et al., 2025b; Wang et al., 2025b), but they rarely verify that an error in one operation changes the success of a dependent operation, or systematically vary these relationships under comparable settings. Without

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.

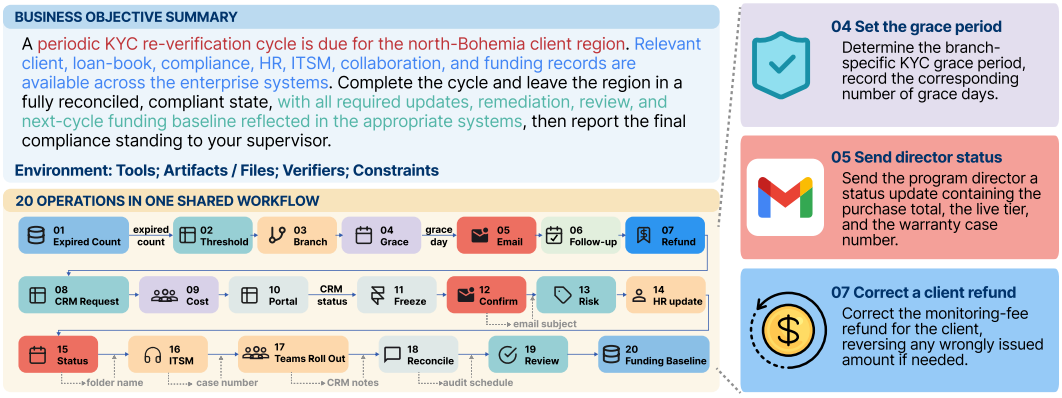

Figure 1: **From familiar enterprise operations to workflow intelligence.** EnterpriseWorld composes the same kinds of operations into a shared executable business process, connecting concrete step queries across systems.

such control, it is unclear how operation-level success relates to workflow completion, or where execution breaks along a dependency chain. We therefore make task dependencies an explicit, verified, and controllable object of evaluation, and build the **EnterpriseFlow** construction pipeline around four design principles: (1) *realistic* workflows grounded in credible enterprise operations and supported by human review; (2) *long-horizon* coordination across many operations; (3) *structural control* over how tasks depend on one another; and (4) *verifiable* local outcomes and task dependencies (Section 3.1).

With these principles, we build **EnterpriseWorld**, a benchmark of 200 long-horizon enterprise workflows of 20 operations each, combining tasks from existing database, enterprise, and application benchmarks with case-specific operations that connect them into a shared executable workflow (Figure 1). Later operations consume earlier results, records, and state changes, often across systems or from several upstream operations, and each workflow's longest required dependency chain (its depth) spans 12–15 operations. These dependencies are verified through testing whether corrupting them changes the success of the dependent operation, and further supported by a human audit with 97.2% valid dependency judgments (Table 3 in Section 4.5). Because dependencies are explicit, they can also be varied: for base workflows, we construct controlled variants that either change how required operations depend on one another or add task/tool distractors while keeping these relationships fixed. Each variant is rerun and revalidated to keep its instructions, dependencies, and targets consistent. Across all settings, agents choose their own execution order, while executable checks assess local outcomes, dependency validity, and workflow completion.

As a diagnostic benchmark that localizes model failures, EnterpriseWorld reveals several distinct limitations of frontier agents. We find that (i) even GPT-6 Astra completes only 46.5% of workflows under our primary completion criterion, with completion dropping by 32–46 points across models as increasingly long portions of the required dependency chain must remain correct; (ii) in ablations, providing the correct intermediate results from earlier operations in the dependency chain improves completion by 22.8–29.5 points for four of five models, while providing the dependency map yields gains of 6.8–10.5 points, and most dependency failures begin when earlier operations produce incorrect results needed later; (iii) agents remain fragile to workflow changes, with 14.0–25.6% of GPT-5.6 Sol's successful base workflows failing under each controlled variant.

Our contributions are threefold. **(1) A Workflow-Centered Construction and Evaluation Framework.** We propose EnterpriseFlow, a framework that treats workflows as *first-class* units of construction and evaluation, making dependencies among enterprise operations explicit, verifiable, and controllable. **(2) A Dependency-Rich Long-Horizon Enterprise Benchmark.** Using EnterpriseFlow, we construct EnterpriseWorld, a long-horizon enterprise benchmark comprising 200 realistic workflows, alongside controlled variants with task/tool distractors and changes in dependency structure. **(3) Diagnosing the Composition Gap in Realistic Enterprise Workflows.** We comprehensively evaluate diverse frontier models (*e.g.*, GPT-6 Astra, Claude Fable 5.1) with process-level diagnostics, revealing that strong local task performance often fails to compose into reliable workflow execution and substantial gaps remain on realistic enterprise workflows.

2 Related Work

Enterprise and long-horizon agent benchmarks. Enterprise operations are a key application of LLM agents, motivating a growing body of benchmarks. WorkArena and WorkArena++ evaluate enterprise knowledge work and compositional planning (Drouin et al., 2024; Boisvert et al., 2025). AppWorld and TheAgentCompany provide multi-application and simulated workplace environments (Trivedi et al., 2024; Xu et al., 2025a). EnterpriseOps-Gym, AutomationBench, and OdysseyBench further evaluate stateful planning, cross-application orchestration, and long-horizon office workflows (Malay et al., 2026; Shepard & Salimans, 2026; Wang et al., 2025b). In contrast, EnterpriseWorld focuses on making the *relationships between operations* a controllable evaluation unit: which upstream results constrain downstream success, how these requirements compose, and how execution changes when their structure is modified.

Executable environment and task construction. EnvScaler scales tool-interactive environments through programmatic synthesis (Song et al., 2026), while Agent-World generates verifiable tasks using tool-dependency graphs and executable programs (Dong et al., 2026). EnvHarness reshapes existing environments through composable changes to state, interaction rules, and task composition (Huang et al., 2026). Our focus is not environment generation or composition alone, but the construction and validation of *critical cross-task dependencies*. EnterpriseWorld weaves source-native operations and case-specific composition tasks into a shared world, jointly generates the workflow graph and live reference computation, and uses counterfactual probes to verify that declared dependencies affect downstream success under reference execution.

Structured evaluation and structural generalization. ToolSandbox uses milestone DAGs to evaluate intermediate progress, and CRAB-Bench uses constraint graphs to construct tasks with interdependent requirements (Lu et al., 2025a; Wang et al., 2026a). AgentNoiseBench instead examines robustness to controllable user and tool noise (Wang et al., 2026b). EnterpriseWorld complements these directions through matched interventions on required dependency paths and prerequisite relations, alongside task and tool distractors that preserve the required graph. These comparisons test whether agents adapt when the dependency topology among familiar operation types changes. Executable node and edge checks further distinguish local correctness, dependency handoffs, and dependency-valid workflow completion, connecting structural variation with process-level diagnosis. In Appendix, Table 4 summarizes these distinctions.

3 EnterpriseWorld: Building Dependency-Rich Enterprise Workflows

An *enterprise workflow* comprises interdependent operations (tasks) that jointly fulfill a business objective. We first describe EnterpriseFlow, the design principles that make these dependencies explicit, verifiable, and controllable for evaluation. Then we introduce EnterpriseWorld benchmark construction, verification, and controlled variation.

3.1 EnterpriseFlow: Workflow Design Principles

In an enterprise workflow, later operations may depend on earlier results, records, or changes to shared business state. Our goal is to evaluate whether agents can coordinate these operations and adapt when workflow requirements or task/tool context change. To support this evaluation, we design EnterpriseFlow around four goals that guide the construction of EnterpriseWorld:

❶ **Realistic.** We compose realistic enterprise operations while preserving their original business semantics and requirements. Sampled human review further assesses workflow and dependency plausibility.

❷ **Long-horizon.** We include chains of dependent operations, tasks that use results produced much earlier, and tasks that require several earlier results together. The goal is to test sustained coordination of results, records, and prerequisites, rather than simply the ability to perform many operations.

❸ **Structurally controllable.** We systematically vary which operations depend on which others (the *dependency topology*) while preserving the business setting and original task requirements. This

enables controlled comparisons of how agents respond when dependency chains or prerequisites change.

❹ **Verifiable.** We use executable checks to assess both individual outcomes and required dependencies. We also test whether replacing a correct earlier result or prerequisite with an incorrect one changes downstream success. These checks help distinguish success at individual operations from execution that satisfies the workflow as a whole.

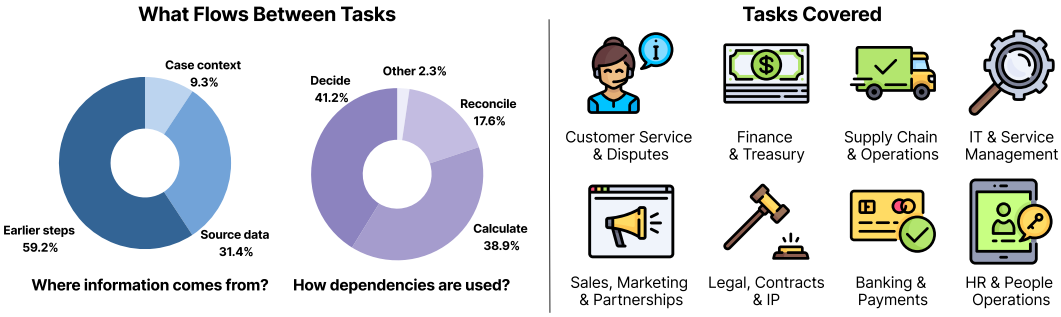

Figure 2: **Dependency composition and task coverage of EnterpriseWorld.** Left: dependencies draw on prior workflow operations, source data, and case context to support downstream operations. Right: the benchmark spans eight enterprise task domains.

3.2 Benchmark Overview

EnterpriseWorld contains 200 base workflows, each comprising 20 business operations (tasks). It combines selected, validated tasks from BIRD (Li et al., 2023), Spider 2.0 (Lei et al., 2025), EnterpriseOps-Gym (Malay et al., 2026), WorkBench (Styles et al., 2024), and AppWorld (Trivedi et al., 2024) with case-specific composition operations. Selected source tasks retain their original data and checks; we call these *source-native operations*. Case-specific *composition operations* connect their results across systems in a shared executable world. Agents receive a business objective, case files, and tools and determine how to complete the workflow through stateful interaction. Figure 2 summarizes where required information comes from, how later operations use it, and the enterprise domains covered by the benchmark.

We therefore represent each workflow as a directed graph $G = (V, E)$, where each node is a business operation and an edge $(u, v) \in E$ indicates that v requires a result, record, or condition from u. The graph is used for construction and evaluation but is not provided to the agent. For example, in a quarterly AP-aging reserve reconciliation, the aged-invoice batch count determines the base reserve, and a `REVIEW` or `ESCALATION` route requires a reviewer handoff email. The credit posting must equal the escalation delta, and the credit ledger must subsequently reconcile to that same delta.

These operations therefore need to use the appropriate upstream quantities and satisfy the applicable prerequisites. Executable checks assess these requirements, while matched topology variants examine how agents respond when dependency paths or prerequisites change. To distinguish operations needed for case closure from other work in the same business event, let $n_f \in V$ denote the case-closing operation. We define the *required operations* and *side operations* as

$$V_{\text{req}} = \{n_f\} \cup \text{Anc}_G(n_f), \qquad V_{\text{side}} = V \setminus V_{\text{req}},$$

where $\text{Anc}_G(n_f)$ contains all operations on which n_f directly or indirectly depends. Side operations lie outside this required dependency flow but remain part of the business event and must also be completed for full workflow completion. Across the 200 base workflows, graphs contain 27.4 dependency edges on average, and the longest required chain has a mean length of 12.9 operations.

3.3 Benchmark Construction and Verification

Figure 3 summarizes the construction and verification pipeline of EnterpriseFlow. More details can be found in Appendix C.2.

❶ **Build a shared executable world.** We place source-native and composition operations in a common business environment. Source data remain separate and read-only, while a writable layer

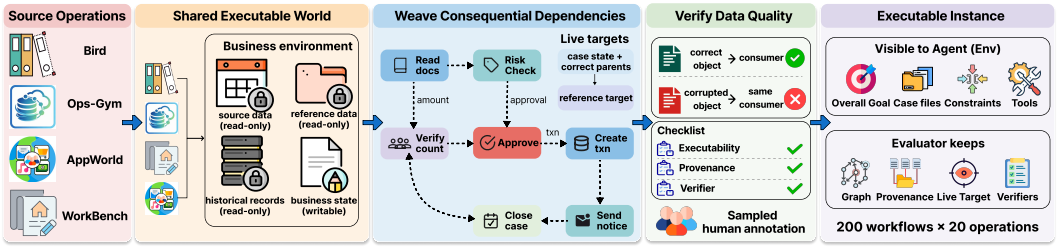

Figure 3: **Constructing EnterpriseWorld with EnterpriseFlow.** Validated source operations and case-specific steps are connected through dependencies in shared executable environment.

supports case-specific records, transactions, messages, and other writable business state. This shared state provides the substrate through which operations from different systems can participate in the same business case.

❷ **Weave consequential dependencies.** We connect operations involving the same business entities or events by requiring downstream tasks to use specific upstream values or records, or to satisfy prerequisites established earlier. We then ground these dependencies in a coherent business event by aligning case-specific entities and writable state, so that connected tasks refer to the same actors, records, and writable process across systems. Each dependency records where the required input comes from, whether preserved through stored results, shared state, or recomputation from its source. Original task checks remain separate from these cross-operation dependency checks.

❸ **Recompute reference targets and verify dependencies.** A reference execution recomputes intermediate outcomes, branch decisions, and final targets from the current case state and correct earlier results. We then test whether each dependency is consequential. For each dependency e, let x_e^* be the correct required input, $\tilde{x}_e$ an incorrect replacement of the same type, and $C_e(x)$ indicate whether the later dependent operation's check passes. We retain the dependency only if

$$C_e(x_e^*) = 1, \qquad C_e(\tilde{x}_e) = 0.$$

For operations with several required inputs, we replace one input at a time.

❹ **Release validated executable workflow.** Before release for human review, we verify task executability and the sources of required inputs, and whether every dependency corresponds to a business requirement available to the agent. Sampled human review further assesses the plausibility and coherence of the resulting workflows. The final workflow exposes only the business objective, case files, constraints, and tools to the agent, while the dependency graph, reference targets, and evaluation checks remain hidden from the agent.

3.4 Controlled Workflow Perturbations

To evaluate agents' robustness to workflow changes, we construct perturbed workflows that modify dependency structure or introduce task/tool distractors while keeping the underlying business setting comparable. An illustration can be seen in Figure 6 (Appendix D). All variants are generated and validated before evaluation; their modified dependencies or added distractors remain fixed throughout each run. The execution environment remains stateful, so agent actions can still change business records and affect subsequent operations. Specifically, we consider two perturbation families:

❶ *Operation dependency perturbations* change how operations depend on one another. (1) Path deepening inserts $k \in \{1, 3, 5\}$ required relay operations along an existing dependency, passing the same required value through additional tasks before it reaches the later dependent operation. (2) Prerequisite addition makes an existing required operation depend on up to three additional earlier operations, without adding new operations. These variants preserve the surrounding business setting while changing the coordination required among its operations.

❷ *Task and tool perturbations* preserve the operation graph but add surrounding context. Task distractors introduce multiple distracting tasks whose triggering conditions are not met in the current case, testing whether the agent avoids incorrectly executing these inapplicable tasks. Tool distractors add multiple irrelevant but callable tools. Let $G' = (V', E')$ denote the operation graph of a matched

variant of $G = (V, E)$. The graph-level changes are

$$\begin{aligned} |V'| &= |V| + k, && \text{for path deepening,} \\ V' &= V, \quad E' \supsetneq E, && \text{for prerequisite addition,} \\ G' &= G, && \text{for task and tool perturbations.} \end{aligned}$$

Path deepening also adds required handoff operations, so differences under this intervention cannot be attributed to topology alone.

4 EXPERIMENTS

Benchmark and Protocol. Our main evaluation covers all 200 base workflows in ENTERPRISEWORLD. Each run starts from an independently reset environment. The agent receives the business objective, case files, and available tools and interacts with the live business state through these tools. After execution, the resulting environment state is evaluated with outcomes and executable checks. Matched variants derived from 100 base workflows are used for the perturbation experiments.

Models and Harness. We evaluate frontier models from OpenAI, Anthropic, and Google. All models use the same agent harness and environment interface, adapted from OpenCode. Appendices C.4 and F provide decoding parameters, execution budgets, evaluation dates, and implementation details.

Workflow-Completion Metrics. Following Section C.1, we distinguish *required* from *side* operations. Our primary metric, *LHCR-Req*, counts a workflow as a success only if the case-closing operation and every operation on which it depends (required operations) pass their executable checks. *LHCR-All* additionally requires all side operations to pass. We also report mean output tokens and tool calls per workflow as measures of execution effort. Additional resource statistics and dependency-level diagnostics are reported in Appendix F.

4.1 CAN AGENTS MAINTAIN CORRECT EXECUTION ACROSS LONG DEPENDENCY CHAINS?

We first examine whether agents can conduct valid execution as increasingly long workflow must remain correct. Figure 4 reports workflow completion and execution effort at increasing dependency depths and over the complete workflow.

Finding ❶ Local errors accumulate into large end-to-end reliability losses. As increasingly long portions of the required dependency chain must remain correct, LHCR-Req falls by 32–46 points from depth 3 to full workflow completion across models. This decline reflects the compounding requirement that upstream operations remain valid throughout the workflow, rather than implying that individual operations intrinsically become harder at greater depth. By depth 10, many workflows have already lost dependency-valid progress, showing how moderately strong local execution can still fail to compose into reliable end-to-end completion. At full depth, LHCR-All closely tracks LHCR-Req, indicating that most failures occur among the required operations for case completion rather than among side operations.

Finding ❷ More tokens and tool calls do not imply better workflow coordination. Token usage and tool interactions increase with dependency depth, yet models that spend more resources do not consistently achieve higher completion. The results suggest that long-horizon failures are not simply resolved by longer or more interactive execution; the central challenge is maintaining valid coordination across dependent operations.

4.2 WHAT PREVENTS AGENTS FROM MAINTAINING CORRECT DEPENDENCIES?

We use targeted oracle interventions to identify which coordination demands most constrain workflow execution. The underlying operations and executable checks remain unchanged, while each oracle removes one source of difficulty. `Procedure Oracle` provides the required procedure and applicable rules upfront. `Dependency-Map Oracle` exposes which upstream operations constrain each downstream operation and the dependency type. `Handoff Oracle` supplies the correct intermediate value, record, or state required by each downstream task, removing errors in producing or carrying the required upstream information. The Procedure Oracle removes both the need to infer the workflow procedure and the delayed discovery of applicable rules. The Handoff

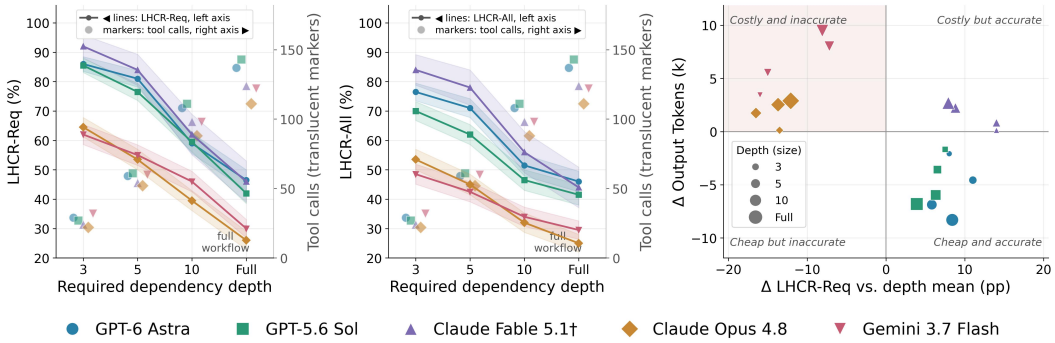

Figure 4: **Workflow completion across longer dependency chains.** Left and middle: cumulative LHCR-Req and LHCR-All as evaluation extends through the required dependency graph, with uncertainty and corresponding tool usage. Right: LHCR-Req versus output tokens at the same evaluation depths. †: Claude Fable 5.1 is evaluated on a 50-workflow subset due to budget constraints.

Table 1: **Ability breakdown with targeted oracles.** Additive oracles selectively reduce planning, dependency-identification, or handoff demands while leaving the underlying business operations and executable checks unchanged.

Model	Setting	LHCR-Req↑	ΔReq.	LHCR-All↑	ΔAll	Tokens (k)↓	Calls↓	Cost ($)↓
GPT-6 Astra	Default	46.5±3.5	–	46.0±3.5	–	12.9±0.21	137±1.8	2.14±0.05
	+ Procedure Oracle	46.0±3.6	-0.5	44.5±3.5	-1.5	12.2±0.12	114±1.1	1.69±0.02
	+ Dependency-Map Oracle	53.3±3.6	+6.8	52.8±3.6	+6.8	11.6±0.12	119±1.3	1.86±0.03
	+ Handoff Oracle	69.8±3.2	+23.3	68.3±3.3	+22.3	11.3±0.11	116±1.3	1.88±0.02
GPT-5.6 Sol	Default	42.0±3.5	–	41.5±3.5	–	14.4±0.19	143±1.7	1.26±0.03
	+ Procedure Oracle	63.5±3.4	+21.5	62.0±3.4	+20.5	13.7±0.18	129±1.6	1.07±0.03
	+ Dependency-Map Oracle	49.7±3.5	+7.7	48.7±3.5	+7.2	13.2±0.16	127±1.4	1.28±0.02
	+ Handoff Oracle	64.8±3.4	+22.8	63.3±3.4	+21.8	12.2±0.14	124±1.3	1.15±0.02
Claude Fable 5.1†	Default	46.0±7.1	–	44.0±7.1	–	23.9±1.01	124±4.6	2.56±0.11
	+ Procedure Oracle	66.0±6.7	+20.0	64.0±6.8	+20.0	18.8±0.59	98±4.5	1.98±0.07
	+ Dependency-Map Oracle	60.0±7.0	+14.0	58.0±7.1	+14.0	20.5±0.60	100±3.6	2.30±0.07
	+ Handoff Oracle	60.0±6.8	+14.0	60.0±6.8	+16.0	18.2±0.68	87±4.2	2.03±0.07
Claude Opus 4.8	Default	26.0±3.1	–	25.0±3.1	–	24.1±0.73	111±2.1	2.02±0.06
	+ Procedure Oracle	48.0±3.6	+22.0	46.5±3.6	+21.5	14.5±0.19	88±1.2	1.16±0.02
	+ Dependency-Map Oracle	34.5±3.4	+8.5	34.5±3.4	+9.5	17.7±0.30	92±1.3	1.54±0.03
	+ Handoff Oracle	51.5±3.6	+25.5	50.0±3.6	+25.0	15.9±0.25	84±1.2	1.29±0.03
Gemini 3.7 Flash	Default	30.0±3.3	–	29.5±3.3	–	30.7±0.44	123±1.5	1.02±0.02
	+ Procedure Oracle	56.5±3.5	+26.5	56.0±3.5	+26.5	24.9±0.28	97±1.1	0.78±0.01
	+ Dependency-Map Oracle	40.5±3.4	+10.5	39.0±3.4	+9.5	29.4±0.35	110±1.2	0.93±0.01
	+ Handoff Oracle	59.5±3.5	+29.5	59.0±3.5	+29.5	24.1±0.34	91±1.2	0.77±0.01

Note: ΔReq. and ΔAll denote absolute percentage-point changes relative to the same model's Default setting. Green/red shading denotes improvement/degradation, with darker shading indicating a larger absolute change. ↑/↓ indicate higher/lower is better;

Oracle provides the correct intermediate value or record when it is needed downstream, so it gains upper-bound errors in both producing and carrying that information. Table 1 reports the results.

Finding ❶ Correct intermediate results matter more than dependency maps for most models. Exposing which operations depend on which others consistently improves completion, but supplying the correct intermediate results produces substantially larger gains for four of five models. Knowing the dependency structure is therefore not enough: agents must also produce and preserve the information required by later operations. Process analysis shows that much of this gain comes from correcting errors made when earlier operations first produce these results, rather than from failures to transfer or retrieve otherwise-correct information. Fable 5.1 (evaluated only on 50 workflows due to budget constraints) remains challenged by EnterpriseWorld and benefits more from procedure and dependency-map assistance, suggesting that its bottlenecks arise earlier in planning and identifying required dependencies.

Finding ❷ Planning remains a major bottleneck for several agents, but not uniformly so. Complete procedure information substantially improves completion for most models while having little effect on GPT-6 Astra, despite reducing its execution effort. This suggests that discovering the required workflow and applicable rules remains an important bottleneck for several models, whereas GPT-6 Astra is less limited by this information. Across models, oracle support also generally reduces token usage and tool calls, suggesting that clearer workflow information can improve execution efficiency even when completion gains differ.

Table 2: **Robustness to controlled workflow perturbations.** Broken-Req and Broken-All report the fraction of Base-successful workflows that fail after the matched perturbation; resource columns report matched changes relative to Base.

Setting	Topo.	Path	Task	Tool	Strength	Broken-Req↓	Broken-All↓	ΔTokens (k)	ΔCalls
Dependency-topology perturbations									
Path deepening ($k = 1$)	✓	✓	–	–	+1 relay	16.3%	19.0%	$+0.82_{\pm 0.27}$	$+2.0_{\pm 2.2}$
Path deepening ($k = 3$)	✓	✓	–	–	+3 relays	25.6%	26.2%	$+1.66_{\pm 0.30}$	$+6.9_{\pm 2.5}$
Path deepening ($k = 5$)	✓	✓	–	–	+5 relays	23.3%	23.8%	$+1.78_{\pm 0.25}$	$+8.7_{\pm 2.0}$
Prerequisite / fan-in	✓	–	–	–	+3 parents	14.0%	11.9%	$+0.36_{\pm 0.28}$	$-0.7_{\pm 2.3}$
Surface perturbations									
Distractor branches (+10)	–	–	✓	–	+10 tasks	18.6%	16.7%	$-0.05_{\pm 0.28}$	$-4.2_{\pm 2.1}$
Tool distractors (+10)	–	–	–	✓	+10 tools	20.9%	19.0%	$+0.15_{\pm 0.26}$	$0.0_{\pm 2.3}$

Note: Parentheses show Base-success→Variant-failure counts. ΔTokens and ΔCalls are paired Variant−Base differences. Configuration columns indicate which aspect is modified relative to Base.

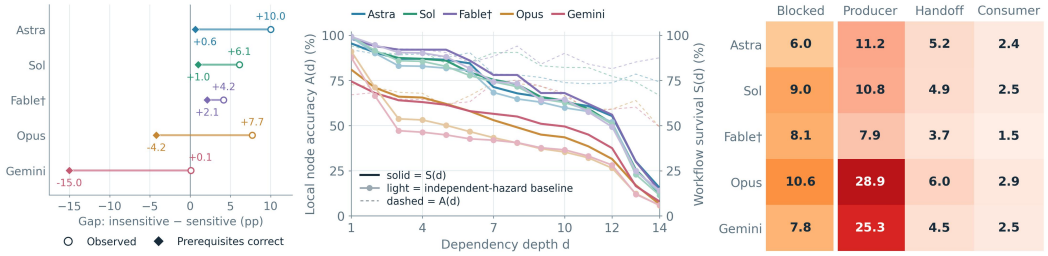

Figure 5: **Process-level analysis of state dependence, long-horizon accumulation, and dependency failures. (a)** State-sensitive performance before and after conditioning on correct prerequisites. **(b)** Local accuracy $A(d)$ and dependency-valid workflow survival $S(d)$ across dependency depth. **(c)** Ancestor-blocked nodes and the first failing dependency stage.

4.3 Can Agents Adapt to Workflow Changes?

We evaluate 100 controlled variants that either change how required operations depend on one another or add task/tool context while preserving these relationships. Because perturbations can both break and recover workflows, we report *Broken-Req* and *Broken-All*: among Base-successful workflows, the fraction that fail after perturbation. We refer to these conditional failure rates as break rates. These break rates measure fragility under the modified setting rather than a pure perturbation effect. Table 2 summarizes the results.

Finding ❶ Successful workflow execution is fragile to controlled changes. Across perturbations, 14–26% of previously successful workflows fail under the modified setting. Structural changes such as path deepening and added prerequisites are associated with substantial loss of previously successful execution, even when the business setting and operation types remain familiar. Since some Base failures also become successful after perturbation, paired break rates expose this loss of previously reliable execution more directly than aggregate LHCR.

Finding ❷ Dependency changes and distractors stress agents differently. Deepening dependency paths consistently increases both token usage and tool interactions, reflecting the additional coordination required by longer dependency chains. By contrast, additional conditional obligations or irrelevant tools barely change execution effort yet still break roughly one fifth of previously successful workflows. Thus, workflow reliability can degrade either because the required dependency structure changes or because irrelevant context makes the same structure harder to execute correctly. Full bidirectional transition counts are reported in Appendix D.

4.4 Process Analysis: Where Does Workflow Coordination Break Down?

Aggregate completion shows whether a workflow succeeds, but not where dependency-valid execution breaks down. We therefore decompose each dependency into three stages: a *producer* creates the required value, record, or state; a *handoff* carries it to the later operation; and a *consumer* uses it. Together with the structural analyses above, this decomposition helps localize failures in state-sensitive, long-horizon execution. Figure 5 summarizes the results.

Table 3: Human validation of sampled dependency annotations. Annotators judge whether the AI verdict for each designed dependency is sound. CIs are Wilson intervals; p is a one-sided exact binomial test.

Question	Human endorses AI verdict	95% CI	p (rate $\leq$ 0.90)
Dependency holds (B1)	97.9%	[94.1, 99.3]	1.8×10^{-4}
Dependency kind (T1)	96.4%	[92.3, 98.3]	1.8×10^{-3}
Provenance re-skin (B2)	100.0%	[74.1, 100.0]	0.31
All dependency questions	97.2%	[94.8, 98.5]	5.4×10^{-7}
Workflow-level (A1–A3)	93.8%	[71.7, 98.9]	0.52

State Sensitivity: Many downstream errors originate earlier in the workflow. More than half of the operations in dataset are state-sensitive, meaning that their correct target, branch, or prerequisite depends on the business state established by earlier operations. Their performance improves substantially when the prerequisite state is correct, indicating that many apparent downstream failures originate upstream; using outdated state is common among the remaining errors.

Long Horizon: Local errors compound across the workflow. Let $A(d)$ denote local accuracy at depth d and $S(d)$ dependency-valid survival through depth d. Under independent local errors, $S_{\text{ind}}(d) = \prod_{j=1}^{d} A(j)$. Local accuracy $A(d)$ remains substantially higher than the fraction of workflows whose required dependency chain remains intact at the same depth. An independence baseline based on the observed local accuracies accounts for much of this gap, suggesting that the decline in end-to-end reliability can largely arise from the accumulation of local errors rather than from an additional depth-specific failure mode.

Failure Localization: Most dependency failures begin at the Producer. The producer–handoff–consumer decomposition further shows that most dependency failures originate at the producer, while handoff and consumer failures are comparatively rare. Once the correct required value or record reaches the consumer, the downstream operation succeeds in nearly all cases. This also clarifies the oracle results above: the Handoff Oracle helps largely because supplying the correct object bypasses upstream production errors, rather than because handoff failures themselves dominate.

4.5 Benchmark Validation and Case Study

Human validation of dependency judgments. We complement the programmatic checks with a sampled human audit in which annotators independently assess designed dependencies and the corresponding AI-judge verdicts. Humans endorse the judge in 97.2% of valid reviews, with 94.7% inter-annotator agreement on double-reviewed cases (Gwet's $\text{AC}_1 = 0.94$). Table 3 provides the full breakdown and statistical analysis.

Case study. Table 15 in Appendix presents a GPT-5.6 Sol execution in which a single upstream misread creates an incorrect state, selects the wrong branch, and propagates into a coherent but invalid downstream workflow. The agent consistently executes and reconciles against its own incorrect state, causing most required operations to fail and yielding $\text{LHCR-Req} = 0$. This case illustrates how a single early error can produce an internally consistent but globally invalid workflow, which operation-level checks alone cannot adequately characterize.

5 Conclusions

We introduced EnterpriseFlow, a framework that treats workflows as first-class units of construction and evaluation through explicit, verifiable, and controllable dependencies. Using this framework, we built EnterpriseWorld, comprising 200 realistic long-horizon enterprise workflows and controlled variants that modify dependencies or introduce task/tool distractors. Our evaluation reveals a composition gap: strong local task performance does not reliably translate into successful workflow execution. Targeted oracles and process-level diagnostics further show that failures can arise at different stages of workflow coordination, while controlled perturbations reveal fragility under workflow changes. Closing this composition gap remains a central challenge for frontier agents to reliably execute enterprise workflows.

ETHICS STATEMENT

Human involvement is limited to the sampled validation audit, and annotators were compensated hourly at standard rates for comparable annotation work. The benchmark release preserves source licenses and provenance and excludes real credentials, private communications, and personally identifying records. Enterprise-agent benchmarks may expose operational capabilities that could be misused, so we recommend sandboxed execution, least-privilege tool access, and checks for unintended side effects. Finally, success on ENTERPRISEWORLD should not be interpreted as evidence that an agent is safe for deployment: the benchmark omits organizational accountability, many legal and social constraints, and the consequences faced by real employees and customers.

REPRODUCIBILITY STATEMENT

Section 3.3 describes workflow construction, dependency verification, and reference-target computation, while Section 3.4 defines the controlled workflow variants. Section C.4 specifies the execution and evaluation protocol, and Section 4 reports the experimental setup, models, and primary metrics. Appendix E provides implementation details on workflow records, source normalization, validation, scoring, perturbation generation, and execution logging. The released artifact will include versioned benchmark instances, source and manifest hashes, exact model snapshots and evaluation settings, raw trajectories, verifier and extractor versions, and the metadata needed to reproduce base–variant comparisons.

AI USE STATEMENT

Generative AI tools were used during manuscript preparation to improve readability, and assist with literature search. Generative AI was not used to determine the executable ground truth or to produce the numerical human-audit results reported in this paper. All substantive claims remain the responsibility of the authors, who reviewed AI-assisted text and checked citations and reported measurements against their primary artifacts.

REFERENCES

Léo Boisvert, Megh Thakkar, Maxime Gasse, Massimo Caccia, Thibault Le Sellier De Chezelles, Quentin Cappart, Nicolas Chapados, Alexandre Lacoste, and Alexandre Drouin. Workarena++: Towards compositional planning and reasoning-based common knowledge work tasks, 2025. URL https://arxiv.org/abs/2407.05291.

Guanting Dong, Junting Lu, Junjie Huang, Wanjun Zhong, Longxiang Liu, Shijue Huang, Zhenyu Li, Yang Zhao, Xiaoshuai Song, Xiaoxi Li, Jiajie Jin, Yutao Zhu, Hanbin Wang, Fangyu Lei, Qinyu Luo, Mingyang Chen, Zehui Chen, Jiazhan Feng, Ji-Rong Wen, and Zhicheng Dou. Agent-world: Scaling real-world environment synthesis for evolving general agent intelligence, 2026. URL https://arxiv.org/abs/2604.18292.

Alexandre Drouin, Maxime Gasse, Massimo Caccia, Issam H. Laradji, Manuel Del Verme, Tom Marty, Léo Boisvert, Megh Thakkar, Quentin Cappart, David Vazquez, Nicolas Chapados, and Alexandre Lacoste. Workarena: How capable are web agents at solving common knowledge work tasks?, 2024. URL https://arxiv.org/abs/2403.07718.

Chengsong Huang, Zifeng Wang, Rujun Han, Jun Yan, Yanfei Chen, Zoey CuiZhu, Ke Jiang, Peng Xia, Han Yu, Yufan Zhuang, Yifei Ming, Jiaqi Pan, Bhavana Dalvi Mishra, Jiaxin Huang, Burak Gokturk, Tomas Pfister, and Chen-Yu Lee. Envharness: Awakening static worlds for agent learning, 2026. URL https://arxiv.org/abs/2608.19880.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues?, 2024. URL https://arxiv.org/abs/2310.06770.

Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, Victor Zhong, Caiming Xiong, Ruoxi Sun, Qian Liu, Sida Wang, and Tao Yu. Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows, 2025. URL https://arxiv.org/abs/2411.07763.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Rongyu Cao, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin C. C. Chang, Fei Huang, Reynold Cheng, and Yongbin Li. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls, 2023. URL https://arxiv.org/abs/2305.03111.

Hongjun Liu, Yifei Ming, Shafiq Joty, and Chen Zhao. Harnessing llm agents with skill programs, 2026. URL https://arxiv.org/abs/2605.17734.

Jiarui Lu, Thomas Holleis, Yizhe Zhang, Bernhard Aumayer, Feng Nan, Felix Bai, Shuang Ma, Shen Ma, Mengyu Li, Guoli Yin, Zirui Wang, and Ruoming Pang. Toolsandbox: A stateful, conversational, interactive evaluation benchmark for llm tool use capabilities, 2025a. URL https://arxiv.org/abs/2408.04682.

Jiarui Lu, Thomas Holleis, Yizhe Zhang, Bernhard Aumayer, Feng Nan, Haoping Bai, Shuang Ma, Shen Ma, Mengyu Li, Guoli Yin, Zirui Wang, and Ruoming Pang. ToolSandbox: A stateful, conversational, interactive evaluation benchmark for LLM tool use capabilities. In Luis Chiruzzo, Alan Ritter, and Lu Wang (eds.), *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 1160–1183, Albuquerque, New Mexico, April 2025b. Association for Computational Linguistics. ISBN 979-8-89176-195-7. doi: 10.18653/v1/2025.findings-naacl.65. URL https://aclanthology.org/2025.findings-naacl.65/.

Chang Ma, Junlei Zhang, Zhihao Zhu, Cheng Yang, Yujiu Yang, Yaohui Jin, Zhenzhong Lan, Lingpeng Kong, and Junxian He. Agentboard: An analytical evaluation board of multi-turn llm agents. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 74325–74362. Curran Associates, Inc., 2024a. doi: 10.52202/079017-2365. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/877b40688e330a0e2a3fc24084208dfa-Paper-Datasets_and_Benchmarks_Track.pdf.

Chang Ma, Junlei Zhang, Zhihao Zhu, Cheng Yang, Yujiu Yang, Yaohui Jin, Zhenzhong Lan, Lingpeng Kong, and Junxian He. Agentboard: An analytical evaluation board of multi-turn llm agents, 2024b. URL https://arxiv.org/abs/2401.13178.

Shiva Krishna Reddy Malay, Shravan Nayak, Jishnu Sethumadhavan Nair, Sagar Davasam, Aman Tiwari, Sathwik Tejaswi Madhusudhan, Sridhar Krishna Nemala, Srinivas Sunkara, and Sai Rajeswar. Enterpriseops-gym: Environments and evaluations for stateful agentic planning and tool use in enterprise settings, 2026. URL https://arxiv.org/abs/2603.13594.

Grégoire Mialon, Clémentine Fourrier, Craig Swift, Thomas Wolf, Yann LeCun, and Thomas Scialom. Gaia: a benchmark for general ai assistants, 2023. URL https://arxiv.org/abs/2311.12983.

Daniel Shepard and Robin Salimans. Automationbench, 2026. URL https://arxiv.org/abs/2604.18934.

Xiaoshuai Song, Haofei Chang, Guanting Dong, Yutao Zhu, Ji-Rong Wen, and Zhicheng Dou. Envscaler: Scaling tool-interactive environments for llm agent via programmatic synthesis, 2026. URL https://arxiv.org/abs/2601.05808.

Olly Styles, Sam Miller, Patricio Cerda-Mardini, Tanaya Guha, Victor Sanchez, and Bertie Vidgen. Workbench: a benchmark dataset for agents in a realistic workplace setting, 2024. URL https://arxiv.org/abs/2405.00823.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. Appworld: A controllable world of apps and people for benchmarking interactive coding agents, 2024. URL https://arxiv.org/abs/2407.18901.

Danqing Wang, Akshay Sivaraman, and Lei Li. Crab-bench: Evaluating llm agents under complex task dependencies and human-aligned user simulation, 2026a. URL https://arxiv.org/abs/2606.01815.

Hongru Wang, Wenyu Huang, Yufei Wang, Yuanhao Xi, Jianqiao Lu, Huan Zhang, Nan Hu, Zeming Liu, Jeff Z. Pan, and Kam-Fai Wong. Rethinking stateful tool use in multi-turn dialogues: Benchmarks and challenges. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 5433–5453, Vienna, Austria, July 2025a. Association for Computational Linguistics. ISBN 979-8-89176-256-5. doi: 10.18653/v1/2025.findings-acl.284. URL https://aclanthology.org/2025.findings-acl.284/.

Ruipeng Wang, Yuxin Chen, Yukai Wang, Chang Wu, Junfeng Fang, Xiaodong Cai, Qi Gu, Hui Su, An Zhang, Xiang Wang, Xunliang Cai, and Tat-Seng Chua. Agentnoisebench: Benchmarking robustness of tool-using llm agents under noisy condition, 2026b. URL https://arxiv.org/abs/2602.11348.

Weixuan Wang, Dongge Han, Daniel Madrigal Diaz, Jin Xu, Victor Rühle, and Saravan Rajmohan. Odysseybench: Evaluating llm agents on long-horizon complex office application workflows, 2025b. URL https://arxiv.org/abs/2508.09124.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments, 2024. URL https://arxiv.org/abs/2404.07972.

Frank F. Xu, Yufan Song, Boxuan Li, Yuxuan Tang, Kritanjali Jain, Mengxue Bao, Zora Z. Wang, Xuhui Zhou, Zhitong Guo, Murong Cao, Mingyang Yang, Hao Yang Lu, Amaad Martin, Zhe Su, Leander Maben, Raj Mehta, Wayne Chi, Lawrence Jang, Yiqing Xie, Shuyan Zhou, and Graham Neubig. Theagentcompany: Benchmarking llm agents on consequential real world tasks, 2025a. URL https://arxiv.org/abs/2412.14161.

Hanwen Xu, Xuyao Huang, Yuzhe Liu, Kai Yu, and Zhijie Deng. Tps-bench: Evaluating ai agents' tool planning & scheduling abilities in compounding tasks, 2025b. URL https://arxiv.org/abs/2511.01527.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering, 2024. URL https://arxiv.org/abs/2405.15793.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023. URL https://arxiv.org/abs/2210.03629.

Ori Yoran, Samuel Joseph Amouyal, Chaitanya Malaviya, Ben Bogin, Ofir Press, and Jonathan Berant. Assistantbench: Can web agents solve realistic and time-consuming tasks?, 2024. URL https://arxiv.org/abs/2407.15711.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. Webarena: A realistic web environment for building autonomous agents, 2024. URL https://arxiv.org/abs/2307.13854.

A A DETAILED COMPARISON WITH RELATED WORKS

A detailed comparison of ENTERPRISEWORLD with related benchmarks and environment frameworks can be seen in Table 4, where we compare among six dimensions.

Table 4: Comparison with closely related approaches.

Work	Enterprise / Workplace	Auto. Construction	Long Horizon	Workflow dependencies	Controlled Reshaping	Topology Intervention
Enterprise and workflow benchmarks						
WorkArena++ (Boisvert et al., 2025)	✓	–	✓	–	–	–
EnterpriseOps-Gym (Malay et al., 2026)	✓	–	✓	–	–	–
AutomationBench (Shepard & Salimans, 2026)	✓	✓†	✓	–	–	–
OdysseyBench (Wang et al., 2025b)	✓	✓†	✓	–	–	–
Environment / Task modification						
EnvScaler (Song et al., 2026)	–	✓	✓	–	–	–
Agent-World (Dong et al., 2026)	–	✓	✓	–	–	–
EnvHarness (Huang et al., 2026)	–	✓†	✓	–	✓	–
Structured task relations						
ToolSandbox (Lu et al., 2025a)	–	–	–	–	–	–
CRAB-Bench (Wang et al., 2026a)	–	✓	–	–	–	–
ENTERPRISEWORLD (ours)	✓	✓	✓	✓	✓	✓

† denotes LLM-assisted construction; **Explicit Structure** names the primary representation connecting task components; **Auto. Construction** denotes systematic task or environment composition; **Controlled Reshaping** modifies an existing task or environment while retaining its underlying identity, whereas **Topology Intervention** specifically modifies required workflow dependencies.

B LIMITATIONS

ENTERPRISEWORLD studies controlled, tool-mediated enterprise workflows and does not capture the full organizational, social, and operational complexity of workplaces. Its perturbations are generated before execution, so they evaluate generalization across changed workflows rather than adaptation to changes that occur during execution. Some structural perturbations also change the amount of required work: path deepening, for example, introduces additional required operations, so its performance effects should not be interpreted as purely structural. The perturbation study is evaluated on GPT-5.6 Sol, so the observed break rates should not be assumed to generalize uniformly across models. Finally, results depend on the evaluated model snapshots, agent harness, and execution budgets, and should not be taken as evidence of real-world deployment readiness.

C MORE DETAILS ABOUT ENTERPRISEFLOW: DESIGNING INTERDEPENDENT ENTERPRISE WORKFLOWS

Given a business goal, a set of tools, and case files, an agent must carry out several related operations in a shared business environment. ENTERPRISEWORLD is a dataset of such workflows in which the relationships between operations are consequential and observable: a later task genuinely depends on an earlier result, and whether that dependency was honored can be checked. This section follows the logic of that design. Section C.1 defines a workflow and its required tasks. Section C.2 describes how we construct dependencies that affect completion. Section C.3 explains how complete outcomes and intermediate progress are observed. Section 3.4 describes how we change a workflow's requirements to test generalization. A procurement case serves as the running example throughout: an agent calculates a payable amount, obtains approval, issues payment, and reconciles the transaction against the ledger.

C.1 WORKFLOW FORMULATION

A workflow is a set of related business tasks carried out in one shared environment. We represent an instance as

$$\mathcal{I} = (s_0, g, \mathcal{T}, \mathcal{F}, G, \Pi, \Phi), \tag{1}$$

where s_0 is the initial world state, g the business goal, $\mathcal{T}$ the available tools, and $\mathcal{F}$ the case files. The workflow graph $G = (V, E)$ records the tasks and their dependencies, Π records where each consumed input originates, and Φ holds the executable checks. The agent sees g, $\mathcal{T}$, and $\mathcal{F}$ and

acts on a live SQLite world through business tools; G, Π, and Φ remain hidden. In the procurement example, s_0 contains the purchase order and the ledger, g asks for the order to be settled and reconciled, and G links calculation, approval, payment, and reconciliation.

A task, a node of G, is one business operation, such as calculating the payable amount or reconciling a transaction. Each task carries a prompt, the relevant tools, declared reads and writes on the world, its source, and a check of its outcome. A dependency, an edge of G, states what one task needs from another: a value, a record identity, an artifact, or a satisfied condition. It also records how that input is handed off, whether it is stored, retrieved, or re-derived, and how far apart the two tasks lie in the reference procedure. These fields let us distinguish a numerical input from an action prerequisite, a distinction that the checks and variants below rely on. Full record schemas appear in Appendix D.1.

The business goal ends in a case-closing action, and this final task determines which work is required. For the final task n_f,

$$V_{\text{req}} = \{n_f\} \cup \text{Anc}_G(n_f), \qquad V_{\text{opt}} = V \setminus V_{\text{req}}. \tag{2}$$

Reconciliation is thus required together with the payment, approval, and calculation on which it depends; we call V_{req} the required closure. Side operations are legitimate side probes or off-chain activities outside this closure, and they let a designed workflow extend beyond its main dependency path. The required closure is the unit that the outcome checks of Section C.3 and the matched variants of Section 3.4 operate on.

C.2 Dependency-Centered Data Construction

Placing several tasks in one database does not by itself create a workflow. The tasks become a workflow only when a later operation actually depends on an earlier result, so that using the wrong value, record, or precondition changes what the later operation achieves. We therefore build each instance around the following design requirements, state the execution demand that each one adds, and close with the pipeline that implements them.

Bringing local tasks into a shared case. Source tasks from query, enterprise-operations, and application benchmarks (Li et al., 2023; Lei et al., 2025; Malay et al., 2026; Trivedi et al., 2024; Styles et al., 2024) are placed in one executable world. Source data are namespaced to retain their identity, and a writable business layer supplies the case's actors, records, and transactions. Composition tasks then connect the source operations into one case. The shared state is what allows an action in one system to matter for work in another: a calculation determines the approval amount, and a payment changes what a later ledger check must find. The added demand is that the agent treat the systems as one case rather than as unrelated queries.

Making the output of one task an input to another. For every consumed input, we identify either its source record or the earlier task that produces it, and the dependency binds the consumer to that particular result. In the running example, payment must use the amount attached to the approved order, and reconciliation must agree with the transaction that payment actually created. The gold procedure computes these bindings from the live case while the workflow is constructed, so every edge has an executable meaning. The added demand is correct value propagation and record selection: a right number attached to the wrong order is a downstream failure, not a near miss.

Sustaining dependencies over a long horizon. We interleave two kinds of dependency with cross-system handoffs, conditional actions, and side tasks. Long-range dependencies return to an earlier result after other work has intervened. Multiple-parent dependencies require several conditions to hold at once: payment needs both the calculated amount and the approval, and reconciliation needs both the payment record and the original obligation. Earlier results remain available through stored artifacts, domain state, or re-derivation, and the edge records which. The added demand is to keep relevant information accessible and to integrate several constraints at the point of use. Each instance records depth, dependency gap, and the number of contributing parents, so these demands can be analyzed separately from task count.

Ensuring that dependencies affect completion. A declared dependency must have a consequence for the downstream check. We replace one upstream witness with a type-valid but incorrect witness and require the consumer's check to change from pass to fail; each parent of a multi-input task is tested separately while the other parents stay correct. A dependency must also correspond to a business requirement that the agent can learn from its instructions or case files, since scoring

sensitivity alone does not establish a necessary execution path. Together these conditions rule out decorative edges whose apparent complexity has no effect on the evaluated task.

Making a final-step shortcut insufficient. The final action is bound to the actual values, artifacts, and conditions produced upstream, and its required closure is checked together with the terminal outcome. A plausible final number therefore cannot substitute for the missing business work. Source records are protected from agent writes, hidden targets are unreachable through the tool interface, and untouched-state, damaged-state, and constant-answer controls test whether missing business work can be concealed by these shortcuts. These safeguards keep completing a case distinguishable from guessing its answer.

Construction overview. The pipeline realizes these requirements in order. It allocates source tasks, materializes the shared world, places local operations, and connects them through a live gold execution; it then probes and strengthens the declared dependencies, renders the agent-visible instructions while hiding evaluation state, validates the instance, and exports synchronized tasks and targets. The eight stages and their admission gates are documented in Appendix E.

C.3 Verifiable Outcomes and Progress

Because a dependency only matters if its violation is visible, each run is observed at three levels. The complete outcome asks whether the final task and its entire required closure succeeded. Local completion asks whether each task's own check passed, regardless of its prerequisites. Prerequisite completeness asks whether a task and all of its ancestors passed. The levels answer different questions about the same run: a locally correct payment record can coexist with a missing approval, and the three views keep such a run distinguishable from one that never paid at all, even though both carry the same overall failure label. Three mechanisms make the underlying judgments trustworthy.

Preserving the meaning of local success. A source task keeps its native judging program, with only the mechanical namespacing needed for the shared world, and cross-task bindings are checked separately as composition requirements. When the business wording and the inherited check disagree, we adapt the wording to the check rather than the check to the narrative. Local completion therefore means what it meant in the source benchmark, while the task's role in the larger workflow is checked explicitly.

Keeping answers aligned with the current case. Expected values are recomputed from the reset state and the correct upstream targets rather than stored as fixed constants. Changing the source data thus changes the reference answer consistently, and a self-consistent chain of agent errors cannot define its own ground truth. This recomputation is also what allows the matched variants of Section 3.4 to carry correct targets.

Checking the report and the business state separately. At the end of a run we freeze the final business state and extract the final report into a separate evaluation table. Report checks and domain-state checks are then applied independently, so an incorrect deliverable is exposed without letting its text repair the business state, and a correct state is credited even when the report misdescribes it.

C.4 Execution and Evaluation Harness

Each run begins from an independent reset of the shared executable environment, with business-state changes persisting across operations. The harness executes agent-selected tool calls and records their effects while leaving operation selection, ordering, retries, and recovery to the agent. Agents receive the business objective, case files, and available tools, while the dependency graph, provenance records, reference targets, and executable checks remain hidden and are used only for evaluation. Business requirements are communicated through the case materials and constraint-delivery protocol: under the *plan-level* condition, constraints are disclosed during interaction when their associated records are first substantively accessed; under the *detailed* condition, the same requirements appear in the initial query. Neither condition reveals reference answers. Appendix E provides implementation details, runtime restrictions, and delivery rules.

The same harness supports dependency-aware evaluation at four complementary levels. *Local task correctness* measures whether individual operations satisfy their executable checks. *Dependency handoffs* assess whether the correct upstream values, records, states, or prerequisites reach the down-

stream operations that require them. *Dependency-valid progress* distinguishes locally successful operations from those supported by a valid upstream dependency closure and tracks this progress across workflow depth. Finally, *workflow completion* evaluates the business-critical dependency closure, with a stricter measure additionally requiring all designed side operations. Scoring combines the terminal business state and final report with tool calls and recorded state changes. This allows the evaluator to check not only what exists at the end of execution, but also whether dependencies and temporal requirements were satisfied during the workflow. Detailed metrics and process diagnostics are defined in Section 4.

D Perturbation Records and Transition Counts

D.1 Workflow Records

One build per workflow. Each workflow is defined by a single executable build module from which every other artifact is derived. The build declares the source databases it merges (`SOURCES`), a small writable overlay of case-specific rows (`OVERLAY`), the 20 operations (`NODES`), a terminal check, an executable final-state verifier, and a reference program `gold(conn)` that carries the workflow out on the live world. Rebuilding an instance must reproduce its committed artifacts byte for byte; Table 5 lists them.

Table 5: **Per-instance artifacts.** Every file is regenerated from the build module; the agent sees only the agent-visible block of `instance.json` and, in the plan-level condition, the plan query and the desk notes delivered at run time.

File	Content	Read by
`build.py`	sources, overlay, 20 nodes with instruction, R/W fields, `depends_on`, per-node verifier SQL; terminal and final-state verifier; `gold(conn)`	harness, gates, perturbation operators
`g_exec.json`	exported executable graph: nodes with role, edges with dependency object, world specification, terminal, final-state verifier, topology summary	scoring, analysis, viewer
`instance.json`	*agent-visible* block: detailed query, recording handbook (`conventions`), sample rows, tool list; *hidden* block: dependency depth, entity bindings, reference trajectory	brief construction
`gt_trace.json`	reference execution trace and `answers_final` (gold report values, 7–12 keys); a record, never a scoring source	extraction whitelist, oracles, variant gate
`overall_query_plan.txt`	the plan-level query: business goal, one line asking the agent to plan the engagement itself, and the verbatim deliverable block	plan-level, map-oracle, and handoff-oracle briefs
`s2_constraints.json`	business-constraint sentences bound to the tables they govern, with trigger type	constraint dispenser
`oracle_map.json`, `oracle_handoff.json`	per-consumer dependency-map and handoff notes for the oracle conditions	constraint dispenser
`perturbations/⟨pid⟩/`	one directory per matched variant: thin loader build, `pert.json` (operator, strength, changed region, gate verdict), variant `g_exec.json` and `instance.json`, plan-level sidecars	perturbation sweep

Node record. An operation n_i carries its instruction, the skills it exercises, the tools it may call, agent-visible context, its input and output artifacts, a field-level read set R_i and write set W_i, an action-semantics tag (reversible, compensatable, or irreversible), its source dataset, a list of executable checks, the identifiers of the operations it depends on, and a role tag (required or side). Read and write sets are field level, either `table.column` or a named value `answers.`$\langle$`kind`$\rangle$; reads and writes are never conflated, because a table-level or read/write-conflated annotation silently corrupts the dependency graph once a workflow has twenty operations. The `answers` table is the shared named-value store through which working figures pass between operations; the agent cannot write it directly (it is fenced at the tool layer), and report values enter it only through extraction at scoring time (Appendix D.3).

Edge record. An edge $n_i \to n_j$ exists only when the producer's write set and the consumer's read set share a field, $W_i \cap R_j \neq \emptyset$. The record names that shared field as the *dependency object*, the actor or entity the object is bound to (*binding*), and the business relationship through which the consumer recovers it (*mode*). Objects that are named values in `answers` are VALUE dependencies; objects that are rows or fields of a business table (a CRM note, a ledger entry, a calendar event, an e-mail) are STATE dependencies. Across the corpus, 55.3% of edges carry a value and 44.7% a state.

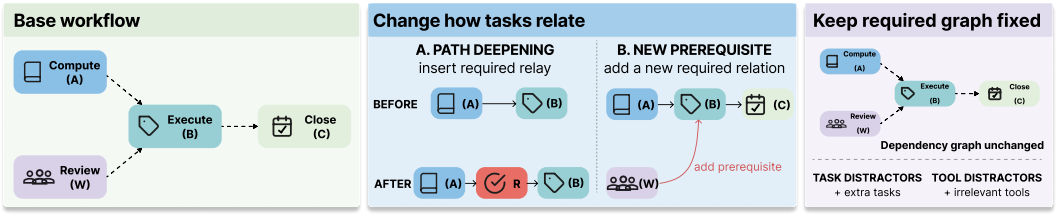

Figure 6: **Controlled changes to workflow structure.** Starting from a validated workflow, EnterpriseWorld modifies required dependency paths by inserting relays or adding new prerequisites. Matched task and tool controls add surface difficulty while preserving the required dependency graph. All variants are recomputed and revalidated before evaluation.

Visible and hidden information. The agent receives the business objective, the case files (the recording handbook and sample rows), and the tool list. The dependency graph, the edge records, every verifier, the reference program, the recorded trace, and the entity bindings are evaluation-side artifacts that the tool surface cannot reach.

D.2 Source Benchmarks, Normalization, and Inherited Checks

Sources and their roles. BIRD-SQL supplies scalar lookups from eleven development databases (financial, superhero, toxicology, formula_1, card_games, student_club, codebase_community, thrombosis_prediction, california_schools, debit_card_specializing, and european_football_2). EnterpriseOps-Gym supplies one oracle task per workflow together with the seed state of the gym servers it touches (calendar, customer service, e-mail, HR, ITSM, Teams, and Drive gyms, often two per task). WorkBench supplies CRM and calendar operations, and AppWorld supplies e-mail, payment, expense-sharing, and task-management operations. Composition operations connect these into one case.

D.3 Live Ground Truth and Scoring

Reference program. The reference program `gold(conn)` is code, not data: it writes each operation's effect and recomputes every figure, branch decision, and downstream target from the live tables, so a change to the initial state or to the graph propagates to every dependent target without relabeling. The recorded trace `gt_trace.json` is a human-readable record of one reference execution and the whitelist of report keys; scoring never reads a target from it.

Negative controls. Every verifier must fail on the untouched world (rule 7); the reference execution must pass every verifier, the terminal check, and the final-state verifier; the final-state verifier must fail under at least three injected damages and under a skip test that omits one early value hop so that downstream completeness fails. A guess gate additionally writes, for each report key, the modal value that key takes across the rest of the corpus (leave-one-out) into an otherwise untouched world and scores every check, measuring how much credit recognizing a house style would collect without doing the work.

Scoring sequence. (1) At run end the harness freezes the world and the final message and scores with the `answers` table fenced; this pre-extraction score is retained for audit only. (2) Extraction parses the deliverable block of the final message, lines of the form *kind: value* after a line reading `REPORT`, keeps only the keys the workflow's gold records, copies the saved world, inserts the parsed values as `answers` rows, and re-runs the unchanged verifiers. Normalization is minimal: whitespace and quotes are trimmed, thousands separators are dropped from pure numbers, and decimals are rendered the way SQLite renders them (15 significant digits), because a model computing in Python otherwise prints 17 digits and is marked wrong for being more precise; this affected 27.4% of finished detailed-condition runs in an early sweep and inverted the model ranking. Everything else is compared as text by the verifiers, so format errors are the agent's errors. (3) Every operation receives its closure verdict. (4) LHCR-Req is 1 when the final operation and all its required ancestors hold their closure verdict, LHCR-All when all twenty do; both are binary per run. Node-level diagnostics (own-check pass, closure pass, ancestor-blocked share) are reported only as diagnostics. After any verifier change, existing traces are replayed on a fresh world and re-scored; models are never re-run to re-verify.

D.4 Validation Gates and Audits

Programmatic gates. Table 6 lists the standing gates a workflow must clear. Each was introduced after a concrete defect that reading alone had missed, and each executes rather than pattern-matches: three earlier corpus-wide counts on this project were wrong by an order of magnitude because they were produced by reading SQL instead of running it. A single command runs the static audit, a pipeline smoke test through a no-op runner, the build self-validation, and an artifact-sync check that rebuilding leaves no repository diff.

Table 6: **Standing validation gates.** All 200 workflows pass every gate at release. Gates marked † propose review items rather than verdicts.

Gate	What it executes	Defect it catches
Build self-validation	edges, graph metrics, rule 7, gold, terminal, final-state verifier on gold and under damage and skip tests	a check that passes on the untouched world; gold that does not satisfy its own checks; a decorative early hop
Verbatim gate	regenerates every inherited recipe and byte-diffs it against the committed check	an inherited check silently edited to fit the narrative
Ground-truth gate	replays the reference program and compares with the recorded trace	stale recorded values copied between workflows
BIRD and seed gates	runs every inherited BIRD gold on its own pinned database; refuses empty or table-less gym seeds	anchors on rows that do not exist; a workflow that cannot run yet looks healthy
Task-cap gate	counts task reuse over the assembled corpus	concurrent authors each seeing a compliant count while the total crosses the cap
Carrier gate	resolves every literal a verifier pins to the goal, handbook, sample rows, or a text column of the world	a rate or string an agent is never shown, unpassable by anyone
Goal-surface gate	finds literals that a verifier requires and a node instruction states but the agent-visible surface never does	three models converging on the same wrong status word
Pinned-id gate†	for every pinned numeric id, checks whether the row exists in the untouched world	an id the agent must create and guess
Derivation gate†	compares goal prose with verifier arithmetic (rounding, filter words, year and day boundaries)	"post-2012" scored as the whole of 2012
Affinity gate	undoes any retyping gold performed and re-scores	a check anchored on gold's own side effect
Guess gate	scores the corpus-modal answer per key on an untouched world	credit for recognizing a house style
Load-bearing probe	overwrites each stored input and re-runs the consumer verifier	decorative dependencies
Binding gate†	classifies each consumer check as operand binding, existence re-assertion, or unreferenced	checks that re-verify upstream instead of consuming it
Static audit	34 dimensions over committed artifacts, including narration truth, compensation binding, verifier tightness, branch source-of-truth, metadata drift, magnitude plausibility, sample fidelity, and dangling `answers` reads	prose that promises a constraint no check enforces; metadata that disagrees with the build

Model-assisted review. Semantic validity is reviewed by a model annotator under a pinned protocol. For each workflow the annotator reads a snapshot from which prior verdicts are stripped, keeps the full 20-step engagement in view, and answers three card types. Card A asks, once per workflow, whether the steps form one workplace case (A1), whether any transition exists only to stitch unrelated steps together with no execution relationship and no ordering a practitioner would recognize (A2), and whether an obvious action, decision, or piece of information is missing between connected steps (A3). Card B asks, for every edge in human scope, whether it makes sense within the engagement for the later step to depend on the highlighted value or state (B1); when B1 holds, which kind of dependency it is (T1: value, state, or precedence); and, for figures that a source question was re-skinned into, whether the original quantity can reasonably represent the workflow interpretation (B2). A branch card asks whether the deciding condition and the differing actions are plausibly related. Human scope is the union of cross-source edges, edges with a composition endpoint, and edges carrying a transformed working figure; same-source internal wiring is judged by the program. Every answer carries a two- to four-sentence reason naming its evidence, and each verdict is stamped with a fingerprint of the cards it was made on so that a later repair cannot leave a stale verdict standing. The wording is pinned because it measurably moved verdicts: on identical cards, asking whether the later step "genuinely needs" the earlier output rejected 7 and 5 edges in two pilot workflows, the schema's wording with the precedence route stated rejected 3 and 2, and the rejected sets were strictly nested; test–retest of the looser wording reproduced only about six in ten rejections, so a single run's rejections are never read as a defect list.

Human audit. Human annotators work in review mode on a web annotation site: each card presents the designed dependency together with the model's verdict and reason, and the annotator marks the

verdict sound or not sound, a comment being required for every not-sound mark and an side note being available for reservations that are not rejections. Cards are presented one at a time with the request, deliverable, and step list in a fixed pane; a guide describes what the program checks versus what a person judges. One attention check per workflow is inserted at a hash-determined position, either an extra question that instructs the annotator to mark it not sound with a given word or a byte-identical repeat of an earlier card, and it is dropped before agreement is computed. The audit reported in Table 3 covers sampled workflows, each reviewed completely by two of six annotators on 2026-09-15 and 2026-09-16; a reviewer's pass enters the statistics only when at least 90% of the workflow's cards are reviewed. Endorsement rates are computed over valid dependency-level reviews, and inter-annotator agreement over cards reviewed by both annotators.

D.5 Annotation Questions

Table 7 lists every question in the annotation protocol of Appendix D.4. A model annotator answers them for each workflow with the full 20-step engagement in view and gives a two- to four-sentence reason for every answer; human annotators then judge whether each verdict is sound, as reported in Table 3.

Table 7: **Annotation questions.** Card A is asked once per workflow, card B once per dependency edge in human scope, and the branch card once per workflow with a conditional branch. Negative answers require a comment; the model additionally gives a reason for every answer.

Card	ID	Question	Answers
A: workflow	A1	Do these steps plausibly belong to the same end-to-end workplace case or objective?	yes / no
A: workflow	A2	Is any transition present only to stitch otherwise unrelated steps together, with no clear business reason? A transition passes on either ground: a real execution relationship, or an ordering a practitioner would recognize as sensible; "nothing is consumed" is not by itself grounds for yes.	no / yes (name the transitions)
A: workflow	A3	Is any obvious action, decision, or piece of information missing between two connected steps?	no / yes (name the transitions)
B: dependency	B1	Within this engagement as a whole, does it make sense for the later step to depend on the highlighted value (for state links: to require the highlighted state) from the earlier step? Judge the business relationship, not whether the instruction says "only after" or "uses"; a link where nothing is consumed but the ordering has a plausible workplace reason is a yes recorded as precedence.	yes / no
B: dependency	T1	If this link is meaningful, what kind of dependency is it? The card's proposed type is shown to confirm or correct.	value / state / precedence (only when B1 = yes)
B: dependency	B2	Can the original source quantity reasonably represent the workflow interpretation in this workflow? Judged on magnitude class, unit, and plausibility of the mapping.	yes / no (only on renamed source values)
BR: branch	B1	Does the condition used to choose among these branches have a plausible business relationship to the actions that differ across branches?	yes / no
BR: branch	B2	Do the actions taken under each branch plausibly follow from that branch decision?	yes / no

D.6 Corpus Statistics

Table 8 summarizes the released corpus. The figures were computed from the committed graph records; the required-depth distribution is 91 workflows at depth 12, 46 at 13, 62 at 14, and 1 at 15.

Table 8: **Corpus statistics.** Means over the 200 workflows unless stated; edge shares are over the 5,490 dependency edges and fan-in shares over the 3,172 consumer operations with at least one incoming edge.

Structure		**Dependencies**	
Workflows	200	Dependency edges per workflow	27.4 (22–42)
Operations per workflow	20	Value / state objects	55.3% / 44.7%
Required / side operations	16.4 / 3.6	Input origin: upstream / native / overlay	59.2 / 31.4 / 9.3%
Max required depth	12.9 (12–15)	Persistence: explicit / implicit / none	55.3 / 41.7 / 3.0%
Reads / writes per workflow	56.8 / 34.5	Edges with gap ≥ 5 / ≥ 10	36.4% / 23.5%
Source databases per workflow	10.9	Consumer fan-in 1 / 2 / 3 / ≥ 4	50.4 / 35.6 / 7.4 / 6.6%
		State-sensitive operations	2,248 of 4,000
Operations		**Verification**	
Reversible / irreversible / compensatable	84.5 / 10.1 / 5.4%	Verbatim inherited checks	7 of 20 ops
Ops using query / create / update / delete	4,000 / 3,506 / 1,203 / 29	Final-state checks per workflow	96 (83–116)
		Workflows passing all gates	200

Note: State-sensitive operations are those whose correct target, applicable branch, or prerequisite depends on an upstream business state: 1,840 state consumers, 624 branch-gated operations, and 197 branch-deciding operations (an operation can be both a consumer and gated). Final-state checks include completeness and collateral checks and vary with the number of guarded arms and compensations in a workflow.

E Execution Protocol

E.1 Runtime

World and agent loop. Before the agent starts, the harness materializes the workflow's SQLite world from its sources and overlay in a private scratch directory, so every run begins from an independent reset. The agent is the OpenCode terminal agent run headless, connected to the world through an MCP bridge that wraps a row-level SQLite backend; OpenCode's native file, shell, web, and sub-agent tools are disabled in the agent configuration, so the six database tools in Table 9 are the only actions reachable. The bridge serializes tool calls through one lock (a model that emits several calls in one turn would otherwise run them concurrently on one connection), appends every call to a ledger, and rebuilds the trace from the ledger when OpenCode's event stream drops events under load.

Table 9: **The agent's tool surface.** Restrictions enforce world integrity, not business compliance; every business requirement is checked only at evaluation.

Tool	**Effect**	**Runtime restriction**
`list_tables`	names of all visible tables	evaluation-side tables (`answers`, reset snapshots) are hidden
`describe_table`	columns and types of one table	as above
`query`	runs one SQL statement	`SELECT`/`WITH` only; any write through `query` is rejected
`create_row`	inserts one row	imported source tables and `answers` reject writes
`update_row`	updates one row selected by a key column	must match exactly one row
`delete_row`	deletes one row selected by a key column	must match exactly one row

Note: A rejected action returns a logged error and changes nothing. The harness never rolls back or repairs an action; retries and compensation are the agent's responsibility and cost only budget.

Budget and termination. A run ends when the agent ends its own turn (*done*) or at a wall-clock cap of 1,800 seconds (*timeout*), at which point the process group is killed and the world is scored as it stands; a provider or startup failure is recorded as *error*. The cap is floored at 1,800 seconds because a 1,200-second cap had silently cut the slowest model and inverted the ranking. A run with zero tool calls that ended in timeout or error is *harness-dead*: the model never acted, and the cell is excluded from the denominator and rerun once the infrastructure is fixed; a real run cut at the cap with work landed is scored and counted as cut. The MCP bridge has a 30-second per-call timeout; the per-run record stores the full trace with arguments and responses, the brief, the final message, per-node check results, usage, wall time, and the transport used.

E.2 BRIEF AND CONDITIONS

System prompt. All conditions share one system prompt:

> You are an operations agent handed ONE business objective. You receive the overall goal, the case files that come with it, and your tools — NOTHING else: no step list, no task breakdown. Decide for yourself everything that must be computed, changed, recorded, sent, scheduled and reconciled to fully satisfy the goal, and do it. All state lives in ONE SQLite database; act ONLY through the provided tools; inspect schemas before writing. Values must be COMPUTED from the live data, never guessed. File your work exactly as the case files' recording handbook says. When the objective is fully satisfied, end your final reply with the deliverable report the goal asks for, then stop.

Brief. The user message has three parts: the user goal, the case files (the recording handbook, which fixes tags, record formats, and the booking rule, and a sample of the initial state), and the tool list. Under the *detailed* condition the goal is the full prescriptive query: every quantity's definition, the policy constants, the branch thresholds, and the step-by-step recipe. Under the *plan-level* condition the goal is the business objective, one sentence asking the agent to plan and carry the engagement out end to end itself, and the verbatim deliverable block that names the report keys; the definitions and the recipe are withheld and arrive as desk notes (Appendix E.3). The world, handbook, tools, cap, and scoring contract are identical, so the difference between the two conditions is what the agent must discover for itself.

E.3 CONSTRAINT DELIVERY

Binding. Each constraint sentence of the detailed query (a quantity definition, a policy constant, a branch rule) is bound offline to the operation whose instruction shares the most content words with it (overlap threshold 0.3) and, through that operation, to the tables its verifier reads. A sentence no operation matches is bound to the start of the run. One additional policy note per workflow, the handbook's booking rule, is bound explicitly to the deepest operation that writes the calendar. Two earlier designs were rejected by measurement: placing the specification in three inbox e-mails made the plan-level condition easier than the detailed one (three labeled documents in a small inbox are a lookup, not a discovery), and dispersing it with version noise and decoy rate cards tested scavenging rather than planning.

Trigger. A bound constraint is delivered the first time the agent *works* one of its tables: a query over that table that shapes the data (a `WHERE`, `JOIN`, or aggregate) or a write to it. Listing tables, describing a schema, or peeking at unfiltered rows delivers nothing, so surveying the world collects no policy; attempting the governed computation is what unlocks it. An earlier trigger on the first touch of a table had collapsed delivery to the start of the run (68% of notes inside the first 10% of calls), because most constraints bind to the source books an agent inspects first. Each constraint is delivered once, in the `desk_notes` field of the tool response, after the triggering action has executed, so the agent must reconcile the requirement with a result it already has; simultaneous deliveries follow a fixed order, and every delivery is logged in the trace with the call that triggered it. A gate checks that no note leaks a gold value, that every graded report key is named by some note, and that notes stay byte-identical across regeneration. The detailed condition never sees a desk note.

Oracle conditions. The two oracles of Section 4.2 ride the same channel. The dependency-map oracle delivers, when the agent first works the tables that mark a consumer operation starting its own step, a note naming which upstream operations constrain that step and the dependency type (value, state, or precedence), never a value or a record. The handoff oracle delivers on the same trigger the correct dependency object as the reference execution produced it: the gold figure of a value dependency or the gold record of a state dependency (up to five candidate rows where attribution is ambiguous). The trigger tables are chosen so that delivery marks the consumer's own step rather than the shared source books, in order of preference the tables only that operation reads, then the tables it writes, then verifier tables read by fewer than three operations; the 11% of consumers that are pure compositions over `answers` receive their note on the first call. A gate proves that every consumer is covered, that no map note contains a gold value, that every handoff value equals the recorded gold, and that every state record is attributed. The procedure oracle is the detailed condition itself.

Perturbation variants. A variant carries its own plan-level sidecars (the base opener, the variant's deliverable block, and the base's desk notes byte-identical), and the harness prefers them when present. For the tool-distractor variant, the decoy tools declared in the variant's record are registered in the MCP bridge as real callable tools with deterministic canned responses, allow-listed in the agent configuration, and ledgered with a decoy flag so that decoy calls are counted per run. A prompt-only tool removal used in an earlier pilot, which the backend still executed, was retired for this reason.

E.4 Operators and Variant Generation

Table 10 specifies the four operators of Section 3.4. A variant is a thin loader over the base build that overrides only the symbols its operator needs and re-exports everything else, so an operator's cost is how many symbols it must override and whether it must write new executable gold. Path deepening is the only operator that writes new gold (the staged records); the prerequisite gate adds composition checks and one handbook line without touching gold; the distractor operator edits the goal and adds absence checks; the tool operator edits only the tool surface. Inherited verifier programs stay byte-identical under every operator.

Table 10: **Perturbation operators.** Each variant applies exactly one operator instance to a base workflow; variants are never derived from other variants.

Operator	What is edited	What stays fixed	Agent-visible change
Path deepening ($k \in \{1, 3, 5\}$)	one load-bearing value edge $p \to c$ into a *required* consumer gains k chained relay operations: the value is re-recorded k times (`X_staged1`, …, `X_staged`k, each copied from the previous) and c re-anchors on the last relay; one composition check per relay (stored value equals its predecessor); gold is wrapped to insert the staged records	c's own value, every other operation, the final task	one handbook line saying the desk stages the value through k records; the report block lists the staged keys
Prerequisite / fan-in (+3 parents)	one *required* consumer v gains three new prerequisite edges $w_i \to v$ from operations not already its ancestors (required parents preferred, acyclicity enforced); one composition check per parent that w_i's evidence is on file when v is	v's inputs and value; gold (every w_i is already executed)	one handbook line stating that v follows the parents
Distractor branches (+10 tasks)	the goal gains ten conditional escalation arms (legal hold, regulator notice, board memo, …), each guarded by a round numeric ceiling at least ten times the case's own figure and any threshold the goal states, so every guard is false; one collateral absence check per arm in the final-state verifier; the operator verifies that firing an arm fails its check	graph, gold, every verifier	the ten arms and their guards in the goal
Tool distractors (+10 tools)	ten decoy tools (SMS, FX rate, geocoding, translation, holidays, reminders, knowledge-base search, CSV export, thread archive, report scheduling) are added to the tool list *and* registered in the MCP bridge with deterministic canned replies; calls are ledgered as decoys	graph, gold, verifiers, the six real tools, the handbook	a wider tool list

Generation-time gate. Every variant is judged from outside before it is marked reliable: rule 7 on the untouched variant world, the reference execution passing every check on the variant, the final-state verifier failing under generic damage, and consistency between the recorded changed region (V_{new}, E_{new}, frontier) and the actual node delta against the base. The verdict is persisted in the variant's record, and an offline audit additionally checks that relays and consumers sit on the required chain, that added report keys are surfaced and whitelisted, and that the plan-level sidecar names every graded key. The 100 sweep workflows were drawn at random from the corpus (seed 20260916); 600 variants were generated and 600 of 600 gated reliable with zero audit failures. The gate operator sits on a required consumer in all 100 workflows; 92 received three new parents, six received two, and two received one, because a required consumer with fewer eligible parents is preferred over an side one with three (a gate off the closure cannot move LHCR-Req). An earlier operator set (single-relay bridges on any edge, one-parent gates, single inactive arms, value seeding, a rule insertion, and a prompt-only tool removal) was retired after a 30-workflow pilot: its relay sat on an side consumer in 157 of 200 workflows, so LHCR-Req could not move by construction, and its deltas were within three points of noise.

Effort under perturbation. Relative to base, path deepening raises mean generated tokens by 0.82k, 1.66k, and 1.78k and tool calls by 2.0, 6.9, and 8.7 for $k = 1, 3, 5$, with every relay recorded in 98–100 of 100 runs; the prerequisite gate changes calls by -0.7 and leaves the consumer carrying a stale base ordering in 9 runs; distractor branches reduce calls by 4.2 and never fire an arm; tool distractors leave calls unchanged (-0.04) while drawing 1.08 decoy calls per run.

F Experimental Details

F.1 Models, Settings, and Accounting

Table 11 lists the evaluated snapshots. All models use the same brief, tools, handbook, cap, sidecars, and scoring; the provider dialect and OpenCode version differ because the models were reached through different transports. Reasoning effort is set to *high* where the provider exposes it (GPT-6 Astra, GPT-5.6 Sol, Claude Fable 5.1); Claude 5-family endpoints reject sampling parameters, so no temperature or top-p is sent to them, and the other models run at their provider defaults. Every cell is one attempt; a completed cell is never rerun to improve it.

Table 11: **Evaluated model snapshots.** Instances are the 200 base workflows unless stated. Dates are in 2026 (Pacific time).

Model	API identifier	Dialect	OpenCode	Effort	Dates	Instances
GPT-6 Astra	`gpt-6-astra`	OpenAI Responses	1.18.31	high	Sep 16–17	200
GPT-5.6 Sol	`gpt-5.6-sol`	OpenAI Responses	1.18.31	high	Sep 16–17	200
Claude Fable 5.1†	`claude-fable-5-1`	Anthropic Messages	1.18.10	high	Sep 17–18	50
Claude Opus 4.8	`claude-opus-4-8`	gateway default	1.18.30	default	Sep 16–17	200
Gemini 3.7 Flash	`gemini-3.7-flash`	gateway default	1.18.30	default	Sep 17–18	200

Note: † evaluated on the fixed 50-workflow subset described below. All sweeps use the same 200-workflow manifest (SHA-256 prefix `caee11bb`). The perturbation sweep of Section 4.3 was run with GPT-5.6 Sol on 2026-09-17.

Accounting. Within a setting the denominator N is the set of workflows any model attempted; a missing run scores 0. Harness-dead cells (Appendix E.1) are excluded from N: nine such cells occurred in the Astra and Sol sweep (an OpenCode startup error after about six seconds), none in the other sweeps. Runs cut at the cap are scored as they stand. Extraction scoring is part of every reported number. Each reported rate is a mean with the bootstrap standard deviation of the mean over 5,000 resamples of the workflow set; output tokens are everything the model generated (output plus reasoning tokens) in thousands per run, and calls are tool calls per run.

The 50-workflow subset. Budget-limited models run on a fixed subset chosen to be representative rather than random: the 200 workflows were stratified by maximum required depth (12 / 13 / 14 or more) and by plan-level difficulty (how many of GPT-6 Astra and GPT-5.6 Sol passed LHCR-Req in base setting: 0 / 1 / 2), with proportional allocation over the nine strata; among 4,000 stratified draws the one kept (seed 20264449) deviates least from the full corpus across all eight Astra and Sol cells (at most 1.5 points). The subset has depth mix 23 / 11 / 16 and difficulty mix 22 / 12 / 16, and 26 of its workflows are also in the perturbation sweep. A subset model is compared with the other models restricted to the same 50 workflows and is never placed in a 200-workflow column.

F.2 Resource Statistics

Table 1 reports execution cost per setting. Prompt tokens include cached prefix reads, which dominate because the whole tool history is re-sent on every step; generated tokens are the quantity reported in the main tables. Wall-clock time depends on the transport and is reported for completeness rather than compared across providers.

F.3 Data and Code

A preview of data and code can be seen at `https://anonymous.4open.science/r/ICLR_EnterpriseWorld_Repo-5E64/`

F.4 DEPENDENCY-LEVEL DIAGNOSTICS

Definitions. All diagnostics are computed from the saved final state and the trace of the plan-level runs. For an operation, the *own-check* verdict is whether its own checks pass; the *closure* verdict additionally requires every dependency ancestor to pass; an operation is *ancestor-blocked* when its own checks pass but its closure fails. Raw and closure node pass rates are the shares of an operation class passing each verdict, averaged over workflows. *Local accuracy* $A(d)$ is the own-check pass rate of required operations at dependency depth d over node-runs; *survival* $S(d)$ is the share of workflows whose required chain is intact to depth at least d, that is, some required operation at depth $\geq d$ holds its closure verdict. For a dependency edge whose consumer the agent reached, the *producer* stage is correct when the producer's own checks pass; *dependency handoff success* (DHS) is the share of edges, among those with a correct producer and a reached consumer, whose binding checks pass; *dependency consumer success* (DCCS) is the share of reached consumers passing their own checks, also reported conditional on a correct handoff. The exclusive first-failure attribution assigns each reached edge to the first stage that fails (producer, handoff, consumer) or to correct. Span is the distance between producer and consumer in the designed order; fan-in is the number of direct parents of the consumer. A state-sensitive failure is classified as *wrong branch* (the agent acted under a non-applicable arm), *stale state* (the producer was correct but the binding failed), *upstream wrong* (the producer was already wrong), or *other local*.

Table 12: **Local accuracy $A(d)$ and survival $S(d)$ by dependency depth.** Percentages; depths 13 and 14 exist in 109 and 63 workflows.

Model	N	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Local accuracy $A(d)$															
GPT-6 Astra	199	92	90	89	90	88	90	79	78	77	74	73	74	79	74
GPT-5.6 Sol	199	94	91	90	91	84	85	90	91	83	82	82	77	74	67
Claude Fable 5.1†	50	95	94	92	92	90	85	88	94	84	90	84	81	85	88
Claude Opus 4.8	200	73	71	68	68	61	67	74	73	72	67	59	59	64	49
Gemini 3.7 Flash	200	67	68	65	64	62	63	73	75	72	68	57	59	60	49
Survival $S(d)$															
GPT-6 Astra	199	95	90	87	87	86	84	71	68	66	63	61	55	30	16
GPT-5.6 Sol	199	99	91	87	87	86	79	75	73	66	64	59	51	26	13
Claude Fable 5.1†	50	98	94	92	92	92	86	78	78	68	68	62	56	30	14
Claude Opus 4.8	200	81	71	66	66	62	58	53	49	45	44	38	32	17	6
Gemini 3.7 Flash	200	74	68	64	63	62	58	56	55	51	50	45	38	16	8

Note: † 50-workflow subset. $S(d)$ is over all N workflows, so it falls at depths 13 and 14 partly because fewer workflows reach them; Figure 5(b) also shows depth normalized by each workflow's maximum.

Table 13: **Handoff success by producer–consumer span and by consumer fan-in.** Percent of edges (span) or of edges and consumers (fan-in), with bootstrap standard deviations over workflows.

	DHS by span					DHS by fan-in			DCCS by fan-in		
Model	1	2–4	5–9	10–14	15+	1	2	3+	1	2	3+
GPT-6 Astra	95.4±0.6	97.2±0.6	97.9±0.7	84.2±2.0	94.3±1.6	97.1±0.4	93.9±0.8	91.1±1.3	95.6±0.5	96.7±0.6	92.6±1.5
GPT-5.6 Sol	95.1±0.6	94.8±0.8	96.2±0.9	91.5±1.2	92.6±2.0	97.4±0.4	94.4±0.8	91.1±1.4	96.3±0.5	94.8±0.7	92.0±1.6
Claude Fable 5.1†	95.7±1.0	96.5±1.4	97.3±1.3	95.3±1.4	93.3±3.3	97.5±0.7	95.3±1.4	95.0±2.1	96.3±0.8	95.2±1.4	95.7±2.0
Claude Opus 4.8	92.3±0.8	92.7±1.1	93.7±1.3	86.9±1.6	89.6±2.7	96.5±0.6	90.2±1.1	87.9±1.8	94.6±0.7	93.4±1.0	89.4±1.8
Gemini 3.7 Flash	93.9±0.7	94.9±0.9	95.9±1.2	92.1±1.4	90.2±2.5	97.6±0.5	93.6±0.9	90.5±1.5	96.1±0.6	94.6±0.8	93.0±1.6

Note: † 50-workflow subset. Span buckets contain 1,245–1,616 / 1,051–1,400 / 443–611 / 669–906 / 174–231 edges per full-corpus model; fan-in buckets 1,116–1,509 / 1,415–1,853 / 1,051–1,400 edges and 1,107–1,474 / 670–883 / 255–338 consumers.

Table 14: **State-sensitive operations.** Own-check pass rates of state-sensitive and state-insensitive operations, before and after conditioning on correct prerequisites, and the exclusive failure categories of failed state-sensitive operations.

	All runs (%)		Prerequisites correct (%)		Failed state-sensitive operations				
Model	Sensitive	Insensitive	Sensitive	Insensitive	Failed	Wrong branch	Stale state	Upstream wrong	Other
GPT-6 Astra	80.6	90.6	94.3	95.0	434	9%	17%	67%	7%
GPT-5.6 Sol	84.4	90.5	93.5	94.5	350	1%	22%	64%	13%
Claude Fable 5.1†	88.6	92.7	94.3	96.4	64	6%	34%	50%	9%
Claude Opus 4.8	66.1	73.9	91.3	87.1	761	15%	11%	68%	6%
Gemini 3.7 Flash	65.2	65.2	93.6	78.6	783	27%	7%	61%	5%

Note: † 50-workflow subset. Among failures whose prerequisites were correct, wrong-branch errors are 0% for every model and stale-state errors are 60–79%. Under the Handoff Oracle both operation classes reach 92–97% and the gap is at most 1.3 points for every model.

F.5 CASE-STUDY TRACE

Table 15: **Case study of a cascading state error.** GPT-5.6 Sol, plan-level condition, balance-transfer workflow (162 tool calls, ended by the agent, reported as "completed and reconciled"). (a) The designed operations in dependency order with the reference outcome and what the agent did; the desk note defining the uplift was delivered at call 47. (b) The deliverable. Green marks an operation or value that passes its check, red one that fails; a red row is nonetheless consistent with the agent's own earlier state.

(a) Execution trace

Step	Operation	Reference	Agent (tool call)	Check
n_{01} [1]	Average loan amount over male-borrower links (`source_balance`)	149,609.18	#52: 149,609.18	✓
n_{02} [2]	Transfer base $= \mathrm{round}(S/200)$, stamped on the CRM as `[BTO-BASE]`	748	748	✓
n_{03} [1]	Promo-window uplift: count of qualifying loans (BIRD `financial` #136, verifier inherited verbatim)	65: every loan meeting the amount, date, and issuance conditions	**Origin.** #53 adds `status='A'`, reading "approved" as a status code → 3	×
n_{04} [3]	Total transfer = base + uplift; delta = uplift	813; 65	751; 3 (consistent with its own uplift)	×
n_{05} [4]	Branch from delta and base against the review floor and the decline bar	`CONDITIONAL_UW_REVIEW`	`AUTO_APPROVE_STANDARD_PROMO`: delta 3 is below the floor	×
n_{06} [4]	Promo-window hold in the enterprise scheduler citing the total	hold cites 813	#113–118: hold cites 751	×
n_{07} opt.	Payoff transfer `[BTO-XFER]` for exactly delta on the card-payoff ledger	65	#107: 3	×
n_{08} [5]	`[BTO-HANDOFF]` e-mail to underwriting, sent iff conditional or declined	sent	not sent (correct for the chosen branch)	×
n_{09} [6]	CRM log `[BTO-LOG]` with the total, only after the handoff	813, after the handoff	751, without a handoff	×
n_{10} [4]	BT-fee allocation on the expense-share ledger from delta	on 65	on 3	×
n_{11} [7]	Follow-up task `[BTO-FUP]` citing delta after the log	65	#111: 3	×
n_{12} [8]	Re-derive the tier from the live book; CRM status `[BTO-TIER]`	`CONDITIONAL_UW_REVIEW`	#123: `AUTO_APPROVE_STANDARD_PROMO`, uplift 3	×
n_{13} [9]	Card-servicing intake ticket gated on tier and base	conditional route	#119–120: auto-approve route	×
n_{14} [10]	`[BTO-STATUS]` e-mail citing total and tier	813, conditional	#110: 751, auto-approve	×
n_{15} [11]	Manager briefing case reassigned, gated on the status e-mail and branch	conditional route	#125–126: auto-approve route	×
n_{16} [5]	Underwriting settle-up matching the recorded share	share of 65	share of 3	×
n_{17} [12]	Wrap-up comment `[BTO-WRAP]` on the follow-up task	total 813	#122: 751	×
n_{18} opt.	Reconcile the ledger net against delta	net 65	#132–162: re-reads its own records, net 3, reports `reconciled: ok`	×
n_{19} [12]	Closing `[BTO-REVIEW]` booking, conflict-free, pinned to uplift, total, delta, and branch	slot after the window; 65 / 813 / 65 / conditional	#112: slot correct; pins fail on 3 / 751 / 3 / auto-approve	×
n_{20} opt.	Quarter-close control reading (`side_scan_count`)	74	74	✓

(b) Deliverable (`REPORT` block)

Key	Gold	Agent	Check
`source_balance`	149,609.18	149,609.18	✓
`transfer_base`	748	748	✓
`side_scan_count`	74	74	✓
`reconciled`	ok	ok	✓
`promo_uplift`	65	3	×
`delta`	65	3	×
`total_transfer`	813	751	×
`branch`	`CONDITIONAL_UW_REVIEW`	`AUTO_APPROVE_STANDARD_PROMO`	×

Note: Bracketed numbers are dependency depths of required operations; opt. marks side operations. The reference for n_{08} is "sent" because the true branch is conditional review; the agent's omission is the correct action for the branch it chose and still fails the check. LHCR-Req $= 0$: 15 of 17 required operations fail their own checks and none is merely ancestor-blocked. Under the Handoff Oracle the correct uplift (65) is supplied at n_{04} and the same workflow passes.

The case study in Table 15 is the GPT-5.6 Sol plan-level run on the balance-transfer workflow (162 tool calls, ended by the agent, reported as "completed and reconciled"). The workflow spans a BIRD reference book, the WorkBench CRM and calendar, Venmo, Splitwise, Gmail, Todoist, and three EnterpriseOps gyms, with a required depth of 12. Two root lookups feed everything downstream: the source balance and the promo-window uplift, from which the transfer base, the

total, the escalation delta, and the branch are derived. At the third substantive query the agent added a status filter (`status = 'A'`) to the count of qualifying loans, reading "approved" as the loan table's status code, whereas the book's own question, inherited verbatim by the verifier and repeated by the desk note delivered at call 47, counts every loan meeting the amount, date, and issuance conditions. The uplift became 3 instead of 65, the delta fell below the conditional-review floor, and the branch flipped from conditional underwriting review to automatic approval. Every gated action that followed was the other route's action: the payoff was wired for 3, the underwriting handoff e-mail that ratifies the payoff was correctly *not* sent for the chosen branch, the CRM log, tier stamp, status e-mail, briefing case, and wrap-up comment all recorded the automatic-approval route, and the closing review booking carried the wrong figures. The agent's reconciliation at calls 132–162 re-derived the chain from its own recorded figures and the ledger it had itself posted, so everything netted and nothing re-asked the book. Fifteen of seventeen required operations fail their own checks (none is merely blocked), four of eight report values are wrong, and LHCR-Req is 0. The same workflow passes under the Handoff Oracle, which supplies the correct uplift at its consumer.