Anchor: Mitigating Artifact Drift in Agent Benchmark Generation

Maksim Ivanov
Agentic Labs
New York, United States
maksim@agenticlabs.com

Abhijay Rana
Agentic Labs
New York, United States
abhijay@agenticlabs.com

Abstract

AI agents are beginning to complete valuable, long-horizon business operations tasks, but training and evaluation environments for enterprise work still struggle to balance realism, verifiability, and scale. Environment and task creation frequently suffers from a failure mode we call *artifact drift*: when instructions, environments, oracles, and verifiers are created by loosely coupled processes, they frequently disagree on what a task requires, producing environments that are unsolvable, reward-hackable, or inconsistent. We introduce Anchor, a task-generation pipeline that formalizes domain experts' specifications of business workflows into constraint optimization programs. From a single parametric specification, the pipeline jointly produces a natural-language instruction, environment configuration, solver-certified ground-truth solution, and state-based verifier. With Anchor, altering parameters yields new tasks with controlled difficulty and known optimal solutions, producing harness-agnostic environments whose rewards depend solely on end-state business correctness. We apply Anchor to produce ERP-Bench: a benchmark of 300 long-horizon tasks spanning procurement and manufacturing workflows in a production-grade ERP system. We find that generation parameters predict realized difficulty, and that frontier models satisfy explicit task constraints in 26.1% of trials but reach a fully optimal solution in only 17.4% of trials. Overall, we show that Anchor and ERP-Bench offer a concrete recipe for building auditable evaluation environments for economically valuable agent work. We release the task generator and ERP-Bench dataset at erpbench.ai.

CCS Concepts

• **Computing methodologies** → **Artificial intelligence**; *Planning and scheduling*; • **Software and its engineering** → *Software testing and debugging*.

Keywords

agent benchmarks, verifiable rewards, enterprise workflows, constraint optimization, ERP systems

1 Introduction

Recent surveys document substantial gaps between benchmark scores and the production performance of language-model agents on enterprise tasks [7, 10, 18]. Audits trace much of this gap to construction errors in the benchmarks themselves. Do-nothing agents pass 38% of τ-bench airline tasks [17] because the verifier accepts empty responses [21], and strengthening unit tests in SWE-bench [6] reranks 40.9% of SWE-bench Lite leaderboard positions [1, 19]. Benchmark authors face an inherent tension between realism, verifiability, and scale. Expert-authored benchmarks improve realism but require costly curation [16]; synthetic generators scale task creation but ship noisy or single-path graders [12, 15]; and τ^2-bench authors observe that earlier benchmarks pushed instructions toward "carefully crafted . . . to help ensure a single, solvable path" [3]. Most benchmarks still author each task's four artifacts of instruction, environment, oracle solution, and verifier in parallel and validate consistency post hoc through audit. This creates a four-way consistency failure we call *artifact drift* (fig. 1): loosely coupled processes end up describing subtly different tasks, such as when the environment lacks data the instruction assumes, the oracle depends on state the environment omits, or the verifier accepts outcomes the instruction did not request.

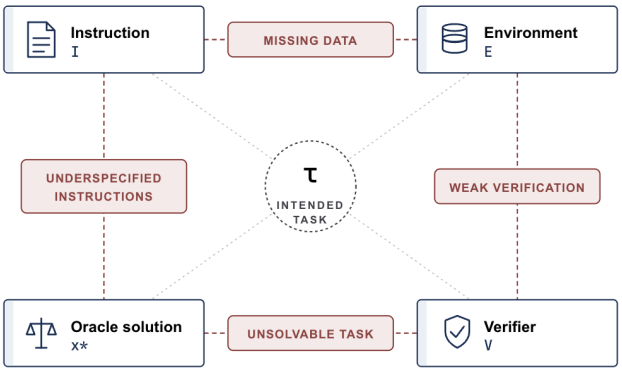

Figure 1: Artifact drift. Inconsistencies between a task's four artifacts—instruction I, environment E, oracle solution x^*, and verifier V—each invalidate the intended task τ in a different way.

We address this with **Anchor**, a task-generation pipeline that compiles all four artifacts from a single solved specification. A domain expert, with an engineer, formalizes a business workflow as a parametric constraint program in OR-Tools CP-SAT [11] with decision variables, business-rule constraints, and an objective metric. For a proposed parameter setting, the CP-SAT solver either rejects the parameters as infeasible or certifies an optimal solution, which then compiles deterministically into the task artifacts. Because all four artifacts are deterministic projections of the same solved specification, the dataset mitigates artifact drift by construction. The same parameters that define a task also tune its difficulty, so the pipeline could support a verifiable training data curriculum for agent training. We apply Anchor to produce **ERP-Bench**, 300 procurement and manufacturing tasks across 29 workflow patterns in Odoo 19, a production-grade open-source ERP system. We evaluate

RLEval '26, San Jose, CA, USA
2026.

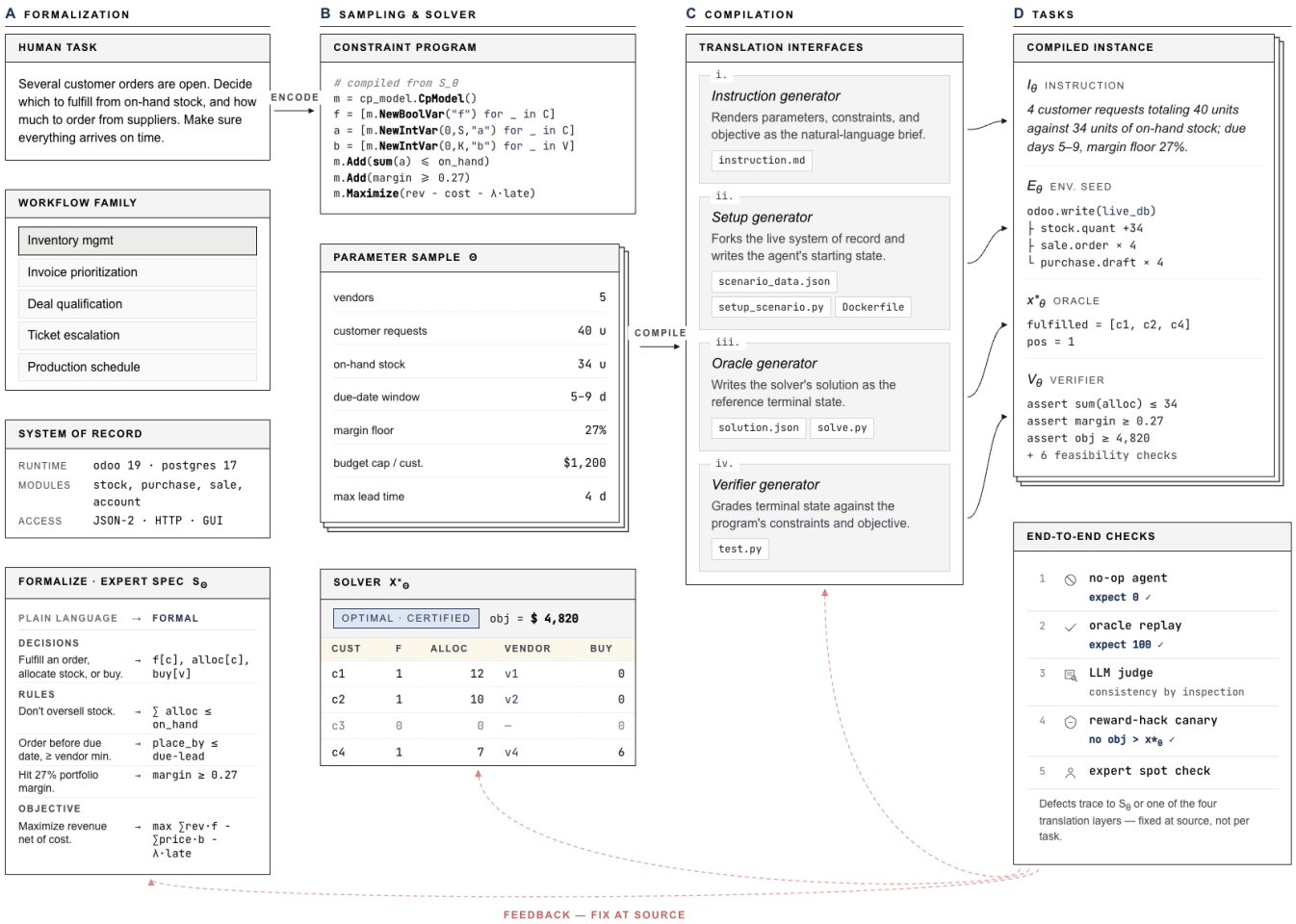

Figure 2: Anchor single-source task creation pipeline. A solved constraint satisfiability problem specification generates the instruction, environment setup, oracle solution, and terminal-state verifier for each task instance.

five frontier models across coding, browser, and computer-use harnesses across 18,000 trials. Pass@5 falls monotonically from easy to hard tiers in every harness, dropping from 70.5% to 22.3% in coding, 46.5% to 7.7% in browser, and 56.0% to 9.5% in computer-use, whith zero instances of reward hacking during evaluation.

Our key contributions are:

- **Anchor**: a task creation pipeline that compiles instruction, environment, oracle, and verifier from one solved constraint program specification.
- **ERP-Bench**: a 300-task verifiable benchmark of long-horizon procurement and manufacturing workflows in a production-grade ERP, with controlled difficulty and certified optima.
- **Evaluation**: a controlled comparison of frontier proprietary and open-weight models across coding, browser, and computer-use harnesses on ERP-Bench tasks.

2 Anchor

Our task creation pipeline follows from prior work in large language models and mathematical reasoning. AlphaProof and related systems treat an informal mathematical statement as the start of a pipeline that translates it into a formal Lean program, where a deterministic checker grades any candidate proof and where synthetic variants of the formalized statement become the curriculum for reinforcement learning [2, 5]. We similarly aim to translate informal business workflows into checkable programs and generate synthetic variants to address the data scarcity and fidelity problem. Unlike AlphaProof and other autoformalization work, we undertake the formalization step manually, and the generated variants are translated back into informal scenarios for agent training and evaluation.

Many enterprise workflows operate on structured data in systems of record, follow explicit business rules, and optimize measurable outcomes, which makes them naturally expressible as constraint satisfiability and optimization problems. The Anchor pipeline (fig. 2) starts when a domain expert and an engineer formalize a workflow such as invoice prioritization, deal qualification, or production scheduling as a parametric constraint program [11] with decision variables, business-rule constraints, and an objective

function. This constraint program becomes the core of the task generation engine.

Given a parameter setting θ, the solver either rejects the sample as infeasible or certifies an optimal solution x^*_θ. We call the resulting parameters, constraints, objective, and certified solution the solved specification S_θ. Four translation layers then compile S_θ into the task artifacts shown in fig. 2: an *instruction generator* renders parameters, constraints, and objective as natural language; a *setup generator* writes the sampled initial records into the environment container; an *oracle generator* writes the solver's solution as the reference terminal state; and a *verifier generator* grades terminal states against the program's constraints and objective. Because the four artifacts are deterministic projections of S_θ, the inconsistencies of fig. 1 are mitigated by construction. Because the solver certifies an optimal objective value, verification remains tractable without contriving the task to a single action path.

Anchor does not eliminate every construction error: the constraint program can encode incomplete business logic and a renderer can mistranslate a correct specification. Five end-to-end checks surface residual defects (appendix I): a no-op agent should score zero on every task, oracle replay should receive full credit, an LLM judge cross-reads artifacts against the CP-SAT program, a reward-hacking canary flags rollouts that beat the solver objective without tripping verifier rules, and domain experts spot-check tasks by hand.

3 ERP-Bench

We apply Anchor to create ERP-Bench: 300 long-horizon procurement and manufacturing tasks across 29 patterns in Odoo 19, an open-source ERP system [9]. Procurement and manufacturing back-office work is economically consequential: manufacturing contributed $2.91T to US GDP in 2024 across roughly 12.6M workers and 239,000 firms [8, 13], and purchasing roles account for 58,700 projected annual openings through 2034 [14]. Mistakes in these workflows directly affect spend, fulfillment, capacity, invoicing, and auditability rather than only surface task completion. Each task runs in its own container against a fresh database seeded with the customers, vendors, inventory, bills of materials, and workcenters the scenario requires. Agents touch the same persistent records a back-office user would, including sales orders, purchase orders, manufacturing orders, vendor pricelists, and invoices, through the JSON-2 API or the standard Odoo web client. For example, a task may ask the agent to fulfill four customer sales orders due within a week when the starting warehouse inventory does not cover them: it must place purchase orders against tiered vendor pricelists that respect minimum-order quantities and lead times, schedule the manufacturing orders that assemble the finished goods from purchased components, link the resulting records back to each sales order while minimizing total spend, and send invoices to the customers with correct payment terms. The 29 workflow patterns were grounded in roughly 40 person-hours of consultation and review with 10 freelance ERP practitioners, and the pipeline then samples each pattern into many task instances as expert effort is incurred per task pattern rather than per instance. The per-pattern roster appears in appendix B, the Harbor task specification in appendix C, and generator details in appendix D.

The verifier combines three dimensions weighted 25/60/15%: *constraint satisfaction* runs discrete checks for demand coverage, deadlines, sourcing rules, manufacturing feasibility, and invoicing; *optimality* compares the realized objective to the certified optimum with exponential decay for suboptimal plans; and *traceability* grades the audit linkage between POs, MOs, invoices, and the sales orders they serve. The constraint dimension gates the others, and a small set of structural prerequisites act as hard zeros (appendix E). ERP-Bench tasks are therefore **both verifiable and open-ended**: the CP-SAT solver certifies an exact optimal objective value and an assignment that achieves it, while agents may reach many valid terminal states through many action sequences.

Difficulty is controlled by parameter groups that compose into easy, medium, and hard recipes. Demand-side parameters scale the number of customers, the size of each order, and how soon each order is due. Supply-side parameters shrink on-hand stock, tighten vendor capacity, and reduce slack between supply and demand. Sourcing parameters progressively unlock more of the ERP surface, layering in single-stage and multi-stage manufacturing, workcenter capacity, and broken initial states that the agent must diagnose and repair. Higher difficulty also enriches the objective, moving from feasibility-only or simple spend objectives toward vendor consolidation, capacity preservation, and plan repair. Appendices B and D summarize the task taxonomy and generation design.

4 Evaluation

We evaluate five frontier models across three harnesses with five trials per agent-task pair, for 18,000 scheduled trials, all sharing one verifier on identical containerized instances. We build the harnesses on the minimal, open-source `pi-mono` agent toolkit [20] so the testbed reflects a real-world agent scaffold rather than a benchmark-specific wrapper used only for evaluation. The coding harness uses shell and filesystem tools, driving Odoo through the JSON-2 API. The browser harness extends `pi` with a Playwright tool that drives the standard Odoo web client through a11y-resolved actions. The computer-use harness drives an Xvfb-backed Chromium through pixel-coordinate clicks, keystrokes, and screenshots, with no DOM access. We evaluate two proprietary models (GPT-5.5, Claude Opus 4.7) and three open-weight models (GLM-5.1, GLM-5V-Turbo, Kimi K2.5). GLM-5.1 is swapped for GLM-5V-Turbo on the computer-use harness because the former does not natively support vision input (appendix F for harness and model details).

The parametric difficulty intervention is the first empirical claim the methodology enables. Across the 300-task release, the declared easy, medium, and hard bands collapse to the same monotone signal in every harness. Aggregate pass@5 falls from 70.5% to 22.3% for coding agents, from 46.5% to 7.7% for browser agents, and from 56.0% to 9.5% for computer-use agents between easy and hard tiers (fig. 3). The strongest negative Spearman correlations against task-level pass@1 come from scale and structure variables such as workcenters, manufacturing orders, solver variables and constraints, BOM component lines, purchase orders, and distinct vendors, while maximum BOM depth and on-hand-stock-to-demand ratio sit near zero because the sampler clamps depth tightly and equalizes stock

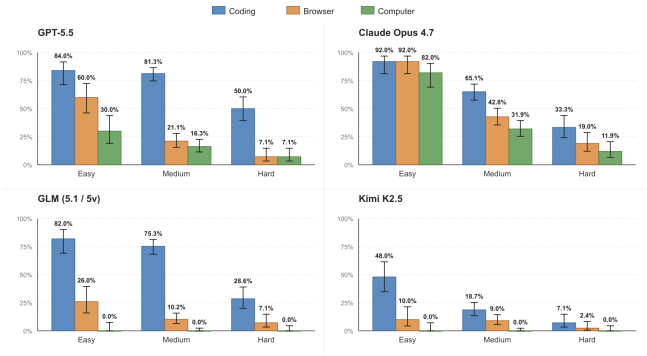

Figure 3: pass@5 by model, harness, and generated difficulty tier (95% Wilson CIs). Generated difficulty tiers correlate with realized performance in every evaluated model and harness.

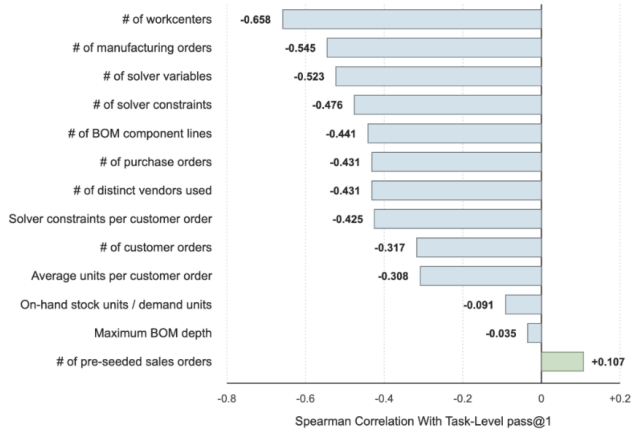

Figure 4: Task parameters correlate with realized difficulty.

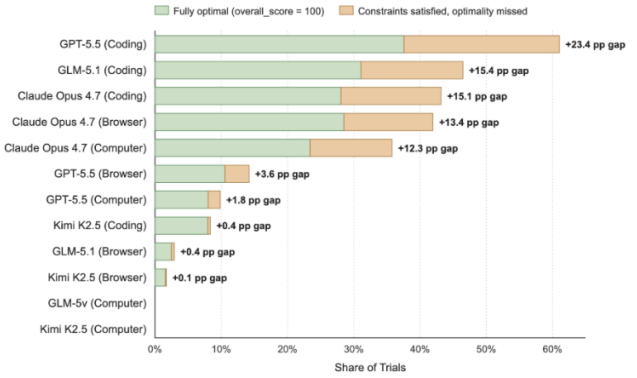

Figure 5: Evaluation reveals a feasibility-optimality gap. Agents satisfy constraints more often than they reach fully optimal solutions.

pressure across difficulty buckets (fig. 4). Task parameters also correlate positively with the number of steps the agent has to take to resolve a task (fig. 6).

The feasibility-optimality gap is the second empirical claim (fig. 5). Across all evaluated models and harnesses, agents satisfy every explicit task constraint in 26.1% of trials but reach a fully optimal solution in only 17.4%, with the strongest evaluated setting, GPT-5.5 in the coding harness, showing a 23.4-point gap. Score loss with difficulty comes from broken constraints rather than worse tradeoffs among feasible plans (fig. 7), so frontier agents remain brittle at business-rule adherence rather than merely choosing more expensive valid plans.

The construction claim holds end-to-end (appendix I): the no-op agent scores zero on every task, oracle replay receives full credit on every task, and no rollout reaches a feasible state strictly better than the certified optimum.

The evaluated models appear more capable and cost-efficient in the coding harness than in browser or computer-use harnesses, even though ERP-Bench is not a coding benchmark and humans complete these tasks through the Odoo GUI. On identical task instances, GUI harnesses resolve 16–56 percentage points fewer tasks than coding harnesses while costing 3.1–14.3× more per attempted task across the two proprietary models evaluated. Appendices G and H report further results, cost drivers, and failure modes.

5 Limitations

The main cost of our task construction is paid upfront in formalization: a domain expert specifies the workflow, an engineer writes a CP-SAT program modelling the entities, constraints, objective, and ERP actions, and a translation layer keeps artifacts aligned. Task validity still depends on the specification. A constraint program can encode an incomplete business rule, a renderer can express the right rule unclearly, and an ERP can expose defaults the formal model did not intend to make relevant. The benchmark covers the part of enterprise work that is naturally visible in terminal system state. Workflows depending on tacit managerial judgment, negotiation, free-text persuasion, or outcomes outside the system of record would not fit well to this framework. Generated instructions are also more explicit than many real workplace requests. ERP-Bench prioritizes feasible, reproducible, and exactly gradable tasks, so released instructions expose the constraints and objectives needed for a fair terminal-state reward.

6 Conclusion

ERP-Bench and Anchor show that using constraint program solvers for agent task generation can be effective in enterprise knowledge work domains. The constructed tasks are realistic, well-specified, and solvable by both humans and agents. Extending Anchor to additional workflows (e.g. sales deal qualification, workforce management, patient intake) and systems of record, (e.g. CRM, HRIS, EHR) for agent evaluation and training are natural next steps.

References

[1] Dena Aleithan et al. 2025. SWE-Bench+: Enhanced Coding Benchmark for LLMs. arXiv:2410.06992 https://arxiv.org/abs/2410.06992
[2] AlphaProof and AlphaGeometry Teams. 2024. AI Achieves Silver-Medal Standard Solving International Mathematical Olympiad Problems. https://deepmind.google/discover/blog/ai-solves-imo-problems-at-silver-medal-level/
[3] Victor Barres et al. 2025. τ^2-Bench: Evaluating Conversational Agents in a Dual-Control Environment. arXiv:2506.07982 https://arxiv.org/abs/2506.07982

[4] Harbor Framework Team. 2026. Task Structure. https://www.harborframework.com/docs/tasks
[5] Thomas Hubert et al. 2026. Olympiad-Level Formal Mathematical Reasoning with Reinforcement Learning. *Nature* 651 (2026), 607–613. doi:10.1038/s41586-025-09833-y
[6] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues?. In *International Conference on Learning Representations*. arXiv:2310.06770 https://openreview.net/forum?id=8y2YPzvJaG
[7] Nikhil Mehta. 2025. Beyond Accuracy: A Multi-Dimensional Framework for Evaluating Enterprise Agentic AI Systems. arXiv:2511.14136 https://arxiv.org/abs/2511.14136
[8] National Association of Manufacturers. 2025. Facts About Manufacturing. https://nam.org/mfgdata/facts-about-manufacturing-expanded/
[9] Odoo S.A. 2026. Odoo 19.0 Documentation. https://www.odoo.com/documentation/19.0/
[10] Alexander Pan et al. 2025. Measuring Agents in Production. arXiv:2512.04123 https://arxiv.org/abs/2512.04123
[11] Laurent Perron and Frederic Didier. 2025. OR-Tools CP-SAT v9.12. https://developers.google.com/optimization/cp/cp_solver
[12] Aarushi Saxena et al. 2025. Continuous Benchmark Generation for Evaluating Enterprise-scale LLM Agents. arXiv:2511.10049 https://arxiv.org/abs/2511.10049
[13] U.S. Bureau of Economic Analysis. 2025. Gross Domestic Product, 4th Quarter and Year 2024, Third Estimate, GDP by Industry, and Corporate Profits. https://www.bea.gov/news/2025/gross-domestic-product-4th-quarter-and-year-2024-third-estimate-gdp-industry-and
[14] U.S. Bureau of Labor Statistics. 2025. Purchasing Managers, Buyers, and Purchasing Agents. https://www.bls.gov/ooh/business-and-financial/purchasing-managers-buyers-and-purchasing-agents.htm
[15] Jian Xie et al. 2026. AgentSynth: Scalable Task Generation for Generalist Computer-Use Agents. arXiv:2506.14205 https://arxiv.org/abs/2506.14205
[16] Frank F. Xu et al. 2025. TheAgentCompany: Benchmarking LLM Agents on Consequential Real World Tasks. arXiv:2412.14161 https://arxiv.org/abs/2412.14161
[17] Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. 2025. τ-bench: A Benchmark for Tool-Agent-User Interaction in Real-World Domains. In *International Conference on Learning Representations*. arXiv:2406.12045 https://openreview.net/forum?id=roNSXZpUDN
[18] Asaf Yehudai et al. 2025. Survey on Evaluation of LLM-Based Agents. arXiv:2503.16416 https://arxiv.org/abs/2503.16416
[19] Cunxi Yu et al. 2025. UTBoost: Rigorous Evaluation of Coding Agents on SWE-Bench. arXiv:2506.09289 https://arxiv.org/abs/2506.09289
[20] Mario Zechner. 2026. Pi Monorepo. https://github.com/badlogic/pi-mono
[21] Mingchen Zhu et al. 2025. Establishing Best Practices for Building Rigorous Agentic Benchmarks. arXiv:2507.02825 https://arxiv.org/abs/2507.02825

Table 1: Domain terms used throughout the paper. Definitions point at the meaning the term carries in Odoo and in ERP-Bench rather than the broader supply-chain literature.

Term	Meaning
ERP	Enterprise resource planning. The system of record a firm uses for sales, purchasing, inventory, manufacturing, and accounting. ERP-Bench uses Odoo.
System of record	The authoritative database for an operational process. ERP-Bench treats the terminal Odoo state as truth.
Sales order (SO)	A customer commitment: who ordered, which product, how many units, at what price, and by when.
Purchase order (PO)	A supplier commitment to deliver goods at an agreed price, quantity, and lead time.
Manufacturing order (MO)	An internal production commitment that consumes components, uses workcenter capacity, and produces a finished good or subassembly.
Bill of materials (BOM)	The recipe for a product. Lists the components and quantities needed to build one unit.
Workcenter	A production resource with limited capacity, such as a machine, station, or line.
Vendor pricelist	A supplier offer for a product, including price, lead time, and minimum order quantity.
Lead time	The delay between placing an order and receiving usable supply.
Minimum order quantity (MOQ)	The smallest amount a supplier will sell under a given offer.
Route	How Odoo supplies a product. ERP-Bench tasks use buy routes, manufacture routes, or both.
On-hand stock	Inventory already available in the warehouse. Agents can allocate it before buying or building.
Origin field	The traceability field on POs and MOs that points back to the SO they serve.
Downpayment	A partial payment required before an order is fully delivered.
Adjacent data	Unrelated business records seeded into the ERP. The verifier checks that agents leave them untouched.
Terminal state	The final ERP database state after the agent finishes. ERP-Bench grades the terminal state, not the trajectory.

Table 2: Buy-and-intake patterns.

Business name	Task count	Planning structure
Routine replenishment cycle	8	Choose POs under stock, MOQ, capacity, price, and lead-time limits.
Buy and bill immediately	8	Confirm accepted SOs, create POs, then issue customer invoices.
Buy with a fixed deposit	8	Add a fixed downpayment invoice before final billing.
Buy with a percentage deposit	8	Compute deposits from retained order value.
Screen a preloaded backlog before buying	10	All candidate SOs are ERP drafts governed by intake rules.
Screen mixed intake before buying	11	Some candidate orders are ERP drafts; others appear only in the brief.
Screen, buy, and invoice	11	Accepted orders require sourcing and the right billing flow.
Lean-context purchasing cycle	10	Adjacent ERP records are sparse; sourcing facts still bind.

Table 3: Disruption-recovery patterns.

Business name	Task count	Planning structure
Supplier cancellation rescue	8	Repair failed POs without an in-house fallback path.
Supplier cancellation rescue with production fallback	10	Rebalance existing POs with new MOs where possible.
Line-outage recovery plan	10	Move seeded MOs to alternate workcenters under reduced capacity.

A Glossary

ERP-Bench borrows operational terminology from enterprise systems. Table 1 collects the domain terms that recur in the paper and the appendix.

B Task Taxonomy

Each ERP-Bench task is an ordinary back-office problem inside Odoo. Customers need products by certain dates; the company has some stock, some suppliers, and sometimes a factory that can build the product or its parts. The agent has to decide what to buy, what to make, which customer requests to keep, and which records to create or fix. A task is solved only when the final ERP state covers the right demand, respects the business rules, and, when applicable, uses the best plan according to the stated objective.

ERP-Bench contains 300 such tasks across 29 workflow patterns generated under fixed seeds. The patterns group into three practical situations: buying goods from suppliers (table 2), repairing a plan after something breaks (table 3), and combining buying with in-house production (table 4). Stable per-task names include the scenario number, difficulty, and generator pattern, so per-pattern result tables are reproducible from directory names.

Objective mix. Some tasks only ask for a valid plan; most also ask for a good plan. In 183 tasks, good means spending the least on new supply. In 34 tasks, it means using fewer vendors; in 31, preserving workcenter capacity; and in 28 repair-plan tasks, changing an existing plan as little as possible. The remaining 24 grade feasibility only. Vendor consolidation, capacity preservation, and repair tasks also use spend as a secondary tie-breaker (appendix E).

Seeded ERP state scale. Each database contains the task-relevant records plus unrelated customers, vendors, products, and business documents for realism. Easy tasks seed roughly 44 customers, 44 vendors, 41 products, and 8 BOMs without any workcenters; medium tasks seed 59, 71, 56, 11, and 1.8 workcenters on average; hard tasks seed 68, 69, 56, 12, and 3.3 workcenters. Across the corpus, 290 of 300 tasks include BOMs in the seeded state, 218 include workcenters, 105 include pre-existing sales orders, 28 include pre-existing purchase orders, and 20 include pre-existing manufacturing orders. Agents must find the records that matter and leave adjacent records untouched.

Product domains and routes. The same planning problem appears in different business contexts, giving agents varied names, catalogs, and surrounding records. Tasks span 12 industrial product categories, most often power equipment (53), industrial equipment (41), office systems (34), material handling (34), building systems (33), safety systems (22), and commercial fixtures (21). Routes determine the available actions: 82 tasks are buy-only, 207 allow both buying and manufacturing, and 11 require manufacturing only.

C Task Technical Specification

ERP-Bench releases each instance as a Harbor task directory. Harbor tasks package an instruction, task metadata, environment files, an optional solution, and tests that emit a verifier reward file [4]. ERP-Bench follows that contract:

Table 4: Production-and-fulfillment patterns.

Business name	Task count	Planning structure
One-line make-or-buy plan	11	Compare finished-good POs against one-BOM internal assembly.
Lowest-cost sourcing route	11	Component prices and finished-good offers both affect spend.
Capacity-smoothing production plan	11	Workcenter limits make the cheapest route infeasible.
Restricted-module manufacturing plan	10	One subassembly has an internal-build requirement.
Two-stage build with cheapest branch	11	The final product depends on one intermediate subassembly.
Two-stage build with qualification rules	11	Only qualified workcenters can build the intermediate part.
Shared-overflow manufacturing plan	11	Backup capacity used by one step reduces capacity for others.
Parallel branch production	10	Independent branches consume separate assigned workcenters.
Sequential build-chain production	10	Later orders depend on outputs from earlier build steps.
Shared-component production	11	Multiple branches consume the same scarce component.
Factory-only by policy	11	Vendor offers exist, but company policy disallows them.
Factory-only by system design	11	Odoo route configuration removes external purchase.
Factory-only by market reality	12	No eligible vendor can supply the finished good.
Screened backlog with vendor consolidation	12	Intake filtering precedes a distinct-vendor objective.
Capacity-smoothing plan with billing	12	Capacity-preserving production also needs invoices.
Screened restricted-module plan	11	Accepted demand depends on a qualified restricted module.
Screened shared-overflow plan with invoicing	11	Retained orders share backup capacity and billing rules.
Screened shared-component plan	11	Accepted orders compete for common intermediate parts.

task/	Harbor task root.
task.toml	Metadata, timeout, and verifier command.
instruction.md	Business brief shown to the agent.
environment/	Odoo build and scenario seed files.
Dockerfile	Container image for Odoo and dependencies.
entrypoint.sh	Service startup and readiness checks.
odoo.conf	Odoo database and module configuration.
setup_scenario.py	Script that creates the sampled ERP state.
scenario_data.json	Generated products, vendors, demand, and constraints.
solution/	Solver-certified oracle replay.
solve.sh	Shell entrypoint for the oracle.
solver.py	Replay code that applies the optimal plan.
optimal_plan.json	CP-SAT objective value and actions.
tests/	Terminal-state grading harness.
test.sh	Test entrypoint run by Harbor.
checks.py	Reward, feasibility, and accounting checks.

Execution semantics. A task starts by loading the sampled company state into Odoo. The agent then creates, updates, confirms, or cancels ERP records through its assigned harness. Evaluation reads the final Odoo database, not the trajectory. The same verifier grades agent runs, oracle replay, and no-op replay, and writes reward artifacts such as the scalar reward, optimality accounting, spend summary, and per-rule results.

D Generator and CP-SAT Specification

D.1 Workflow Formalization

Ten freelance ERP practitioners with five or more years of production experience contributed roughly forty person-hours of elicitation and review across the 29 workflow patterns. An engineer then encoded each pattern as a parametric CP-SAT program. For `09_single_bom_lowest_cost`, the integer decision variables are:

$q_{vp} \geq 0$	units purchased on offer v, p,
$b_{vp} \in \{0, 1\}$	indicator that offer v, p is used,
$a_w \geq 0$	units assembled on route w,
$s_i \geq 0$	stock allocated to order i.

Constraints enforce per-order demand coverage from stock, purchases, and assembly; MOQ-and-capacity tiers $L_{vp} b_{vp} \leq q_{vp} \leq U_{vp} b_{vp}$; BOM feasibility so each assembled unit draws on-hand or purchased components; workcenter capacity; and on-time delivery against per-order deadlines. The objective for this pattern is minimum new spend across confirmed purchases and assembly. Other patterns reuse the same feasibility region: vendor consolidation minimizes used offers, capacity preservation minimizes scheduled workcenter minutes, repair tasks minimize distance from a seeded baseline plan, and feasibility-only tasks omit the optimization phase.

CP-SAT [11] fits this construction because it is a portfolio solver that pairs SAT-style search with linear-programming relaxation. The b_{vp} channeling, BOM disjunctions, and cumulative workcenter capacity each have native primitives in CP-SAT, where a generic MIP formulation would need big-M expansions for the on-off conditions or an $O(n \cdot T)$ time-indexed expansion for the cumulative constraint. CP-SAT also returns a certificate of optimality, which the verifier uses downstream to grade realized terminal states against a known optimum.

D.2 Difficulty Recipes

Difficulty is controlled by recipe parameters that scale demand, deadlines, stock, vendor capacity, manufacturing depth, workcenter capacity, intake screening, invoicing, and repair disruptions. Authored configurations vary sourcing structure, capacity regime, order-acceptance policy, objective type, and industrial domain. Table 5 reports representative ranges for each tier.

Within those ranges, the sampler draws concrete instances from a small set of distributions: discrete uniforms for customer counts, deadlines, demand quantities, vendor counts, lead times, and MOQ tiers; continuous uniforms for stock ratios, vendor-capacity ratios, required margins, and budget multipliers; Gaussian noise around catalog list and standard prices for per-customer prices and per-vendor costs; and categorical draws for product domain and vendor-category mix. A seeded `numpy` generator makes every draw reproducible.

This parameter diversity is the lever for solution diversity. Different sampled vendor prices, MOQ tiers, and capacities push the CP-SAT optimum onto different vendor subsets. Different stock ratios shift which orders are covered from stock, purchases, or assembly. Different component prices flip individual products between make and buy, and different deadline draws change which supply paths remain on time. Two tasks generated from the same authored pattern can therefore force qualitatively different decisions despite sharing a constraint family and an objective shape.

Recipes compose along independent axes: a medium task can stay buy-only with intake screening and invoicing, or add manufacturing; a hard task can combine tight supply with multi-stage BOMs, shared or qualified workcenters, or a seeded disruption.

Table 5: Representative generator recipes by difficulty. Ranges are sampled uniformly unless the scenario fixes a value. Stock ratio and vendor capacity ratio are relative to total sampled demand.

Difficulty	Customers	Demand	Stock ratio	Vendor cap ratio	Tightness	BOM structure	Workcenters	Objective
easy	[4, 4]	[1, 11]	[0.75, 0.92]	[0.40, 0.90]	0.25	none	0	feasibility or spend
medium	[8, 10]	[14, 25]	[0.38, 0.52]	[0.10, 0.36]	0.55	none or single BOM	0 or 3	spend or vendor count
hard	[10, 32]	[15, 31]	[0.04, 0.42]	[0.07, 0.26]	0.62–0.72	multi-stage BOM, optional repair	3	capacity or repair

Table 6: Solver status accounting across generation phases.

Phase	Calls	Optimal	Infeasible	Unknown
Feasibility	24	24	0	0
Primary objective	1,324	1,120	204	0
Primary retry	204	0	204	0
Spend secondary	468	468	0	0
Lexicographic fixed-search	1,144	1,113	0	31
Fixed-search retry	31	3	0	28
Total	3,195	2,728	408	59

Table 7: Solver performance and CP-SAT model size by difficulty tier. Solve-time columns report cumulative wall time per accepted task, including failed and retry solver calls and the tie-breaker solve.

Difficulty	Tasks	Avg solve	Median	P95	Max	Vars	Discards	Resamples
easy	50	0.044s	0.008s	0.288s	0.347s	19.8	253	20
medium	166	0.951s	0.031s	2.842s	20.863s	78.6	243	79
hard	84	5.856s	0.100s	31.984s	100.735s	96.7	236	56

D.3 Sampling, Rejection, and Determinism

Anchor generated 300 accepted tasks after rejecting 732 sampled parameter sets, in 656.76 s of wall time. Pre-solver discards (275) catch cheap arithmetic failures such as demand exceeding total available supply; solver discards (457) catch parameterizations CP-SAT proves infeasible after a 5-second attempt, optionally retried at 15 seconds. Among successful solves, a multi-phase sequence first finds a feasible plan, optimizes the primary objective, optionally optimizes a spend secondary, and then runs a fixed lexicographic search that selects one stable plan among equal optima so that the instruction, environment, oracle, and verifier all refer to the same plan. Table 6 reports the per-phase call counts; the 59 `UNKNOWN` statuses sit in tie-break phases and leave the certified primary solution intact.

D.4 Solver Performance by Difficulty

Table 7 reports cumulative solve time per accepted task, including failed samples, retry solver calls, and tie-breaker solves. Average solve time rises from 0.044 seconds on easy tasks to 0.951 seconds on medium tasks and 5.856 seconds on hard tasks. The increase tracks CP-SAT model size, from 19.8 variables on easy tasks to 78.6 on medium tasks and 96.7 on hard tasks, as recipes turn on manufacturing, workcenter capacity, screening, and repair constraints. The P95 and maximum columns show why a short initial budget is not enough for every sample: hard tasks include a long tail of difficult parameter settings, which the generator resolves by retrying or resampling before release.

E Verifier and Reward Details

E.1 Verifier Protocol and Reward Formula

The verifier is a terminal-state coprocess. A shell script waits for the seeded Odoo to come up, then feeds named check invocations over stdin to a Python checker that returns PASS, FAIL, or NA on stdout. NA marks rules that do not apply to the task, so per-task rule density reflects only checks that scored the run. The same script grades agent rollouts, oracle replay, and the no-op baseline, and writes per-rule, spend, optimality, and aggregate-reward artifacts under `/logs/verifier` for downstream auditing.

Let c and t be the per-dimension percent of applicable checks passed for the constraint and traceability dimensions, and o the optimality score (appendix E.3). The final reward is

$$R = \begin{cases} 0 & \text{if a hard-zero gate fires,} \\ 0.25\,c & \text{else if } c < 100, \\ 0.25\,c + 0.60\,o + 0.15\,t & \text{otherwise.} \end{cases}$$

Hard-zero gates are defined in appendix E.4. The constraint gate ensures an agent cannot trade an explicit business rule for a cheaper plan.

E.2 Rule Catalog

Constraint and traceability checks come from a fixed catalog grouped into six functional families (table 8). Each task instantiates the applicable subset with sampled arguments. Demand and sales dominates because every task begins with customer demand: each customer-product pair drives a separate coverage, deadline, list-price, revenue, and budget check. The remaining families enter only when the sampled pattern requires manufacturing, screening, invoicing, or repair logic.

E.3 Optimality Calculation

The verifier recomputes the realized primary objective a from terminal Odoo records and compares it with the certified value e. The four objectives with an optimization phase share the exponential decay

$$\texttt{score}(a, e) = \begin{cases} 100 & \text{if } a \le e + \tau, \\ 100 \cdot \exp\big(-k\,(a - e)/\max(e, 1)\big) & \text{otherwise,} \end{cases}$$

with per-objective τ and k in table 9. Objectives with a secondary spend metric combine primary score p and secondary score s lexicographically with band weight $w = 0.1$: $o = (1-w)\,p + w\,s$ when $p \ge 100$, and $(1-w)\,p$ otherwise. The secondary can lift a fully satisfied primary above 90 but never compensates for a primary regression.

Table 8: Rule families with representative checks and scored invocations across the 300-task release. The five constraint families gate the optimality and traceability slices of the reward (appendix E.3); the traceability family is graded into the traceability slice directly.

Rule family	Representative checks	Scored
Demand and sales	demand_coverage, deadline_fulfillment, list_price, sale_revenue, budget_compliance	15,933
Supply and procurement	supply_timing_feasible, supply_coverage, po_price_tier_compliance, po_min_qty_compliance, po_consolidation_compliance, new_spend_margin_policy	2,840
Invoicing and payment	regular_invoice_amount_matches_policy, downpayment_invoice_amount_matches_policy, rejected_order_not_invoiced	2,132
Traceability and adjacency	po_origin_traceability, mrp_origin_traceability, linked_posted_invoices_are_tax_free, adjacent_data_untouched	1,935
Manufacturing and capacity	mo_schedule_compliance, mo_component_feasibility, assembly_capacity_compliance, forbidden_finished_mo_absent	1,404
State and seeded orders	task_state_transitions_completed, seeded_order_confirmed, seeded_order_cancelled, repair_state_compliance	976

Table 9: Optimality objectives and decay parameters. τ is the zero-penalty tolerance; k controls how quickly the score decays as the realized metric overshoots the oracle.

Objective	Primary metric a	τ	k	Secondary
min_new_spend	Realized new spend across POs and MOs	0.25	5.0	none
vendor_consolidation	Distinct relevant vendors used	0	2.0	spend
capacity_preservation	Scheduled workcenter minutes	0.01	5.0	spend
repair_plan	L_1 distance from seeded baseline plan	0	2.0	spend
constraint_only	not applicable	—	—	fixed at 100

E.4 Hard-Zero Gates

Two structural conditions collapse the reward to zero even when individual constraint checks pass. A *partial-acceptance* gate fires on screened tasks when the agent leaves every task-relevant record untouched, and a *repair-state* gate fires on repair-plan tasks when the seeded broken plan is still present in the terminal state. The gates exist because screened tasks reward an explicit triage decision a no-op never demonstrates, and repair tasks measure deviation from a baseline an unmodified state has zero of; without the gates both task types would reward inactivity.

E.5 Reward-Hacking Defense

The agent has write access to the PO price_unit field, so a naive verifier could be tricked into rewarding a small price entered by hand. The optimality dimension defends against this by re-pricing each PO line from the authoritative vendor offer tier on file rather than reading the agent-written field, and the tier-price compliance check in the constraint dimension fails the line if it deviates from the tier price by more than a small tolerance. A failed tier check clips the score through the constraint gate, so an agent that writes a wrong price loses constraint credit regardless of optimality.

E.6 Rule Density

Table 10 reports the per-task count of scored verifier checks. Hard tasks instantiate more independent rules because they combine manufacturing, sourcing, capacity, and repair constraints in a single scenario.

F Harness Details

We choose the three harnesses to separate task competence from interface. Coding tests direct API automation, a natural scaffold for terminal-oriented agents; browser and computer-use test less integrated UI modes closer to human Odoo work. All three share the same pi CLI, task image, seeded database, instruction, and verifier; only the pi-mono adapter and tool surface change.

Table 10: Scored verifier checks per task by difficulty tier. The denominator counts applicable checks only; NA returns stay out of the rule density numbers.

Difficulty	Tasks	Avg rules	Median	Range
easy	50	55.76	63	38–66
medium	166	73.01	76	29–146
hard	84	122.76	105	66–208
overall	300	84.07	76	29–208

Table 11: Harness implementation surfaces. Adapters live under agents/. UI extensions mask the default coding tools, so browser and computer-use agents cannot call shell, filesystem, or API helpers directly.

Harness	Adapter / extension	Tool surface visible to the agent
Coding	pi.py; default pi configuration	read, write, edit, bash, grep, find, and ls; Odoo effects go through JSON-2 API calls.
Browser	pi_browser_use.py; browser extension	one browser tool; up to 7 accessibility-tree actions per call.
Computer-use	pi_computer_use.py; computer-use extension	one computer tool; up to 16 mouse, keyboard, or screenshot actions per call.

The full evaluation schedules 5 trials per agent-task pair. Each trial starts from scratch with no retries, a one-hour timeout, a 400-turn budget, and provider-default reasoning effort where exposed. UI containers add only the packages needed to expose the interface, including Xvfb and Chromium for computer-use trials. The computer-use runs for open-weight models were halted early after over 500 trials produced zero points, and their results reflect partial trials.

Model access. Proprietary models are accessed via first-party developer APIs; open-weight models are accessed via OpenRouter.

G Additional Results

G.1 Headline Metrics by Model and Harness

Table 12 reports evaluation metrics for each model and harness over the 300-task release. Every model-harness pair schedules 1,500 trials (300 tasks × 5 trials); the two halted computer-use runs stopped early after producing zero points across hundreds of attempts. Per-pattern pass@5 across the 29 task patterns ships with the dataset release.

The cost of moving from coding to a UI harness is uneven across models. Pass@5 falls by 49 percentage points for GPT-5.5 and 51 for GLM-5.1 from coding to browser, and by 56 for GPT-5.5 from coding to computer-use. Claude Opus 4.7 loses 16 and 22 percentage

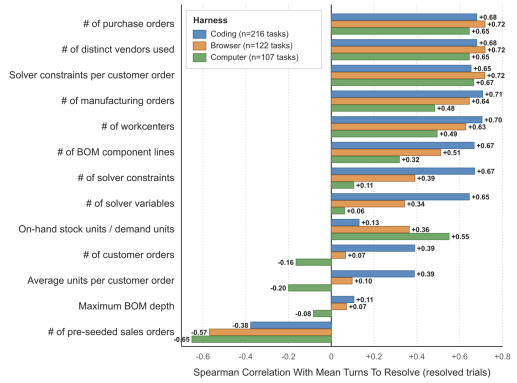

Figure 6: Spearman correlations between task parameters and mean turns to resolve, on resolved trials, reported per harness.

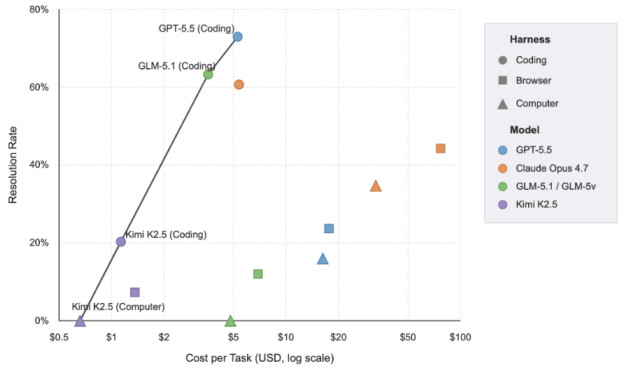

Figure 8: Resolution rate against dollar cost per task. The pareto frontier is traced by Kimi K2.5, GLM-5.1, and GPT-5.5 in the coding harness.

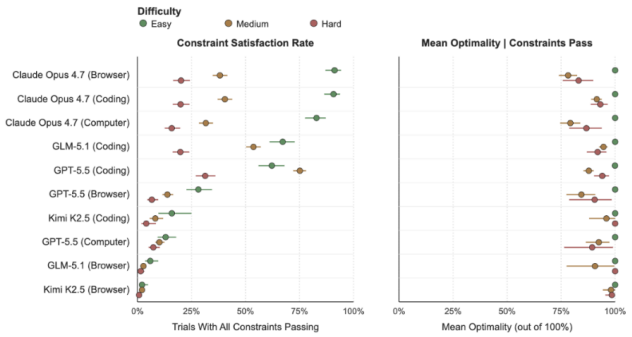

Figure 7: Constraint satisfaction and mean optimality given constraints pass, broken out by model, harness, and difficulty tier.

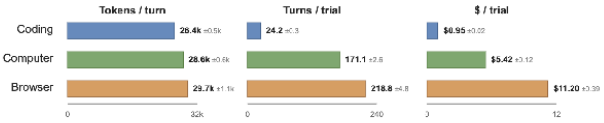

Figure 9: Per-harness cost drivers. Tokens per turn are comparable across harnesses; turns per trial drives the cost gap.

points across the same two transitions, the smallest GUI penalty of any evaluated model.

G.2 Task Parameters Predict Action Burden

Generated difficulty predicts how much work success takes, not only whether success occurs. Figure 6 reports per-harness Spearman correlations between task parameters and turns to resolve, computed on resolved trials. The scale variables that lower pass@1 in fig. 4—purchase orders, distinct vendors, manufacturing orders, components, BOM lines—raise turns to resolve with correlations of +0.6 to +0.7 in every harness.

G.3 Feasibility and Optimality Split

Figure 7 breaks the feasibility-optimality gap of fig. 5 down by model, harness, and difficulty tier. Constraint satisfaction collapses with difficulty across the board, but mean optimality among constraint-clean trials stays above 86 wherever at least one clean trial exists. Score loss with difficulty comes from broken explicit constraints, not from worse trade-offs among feasible plans.

G.4 Cost and Step Efficiency

Figure 8 plots resolution rate against dollar cost per task. The pareto frontier is traced by three coding points (Kimi K2.5, GLM-5.1, and GPT-5.5); every UI point and the Claude coding point sit below it. Moving the same model from coding to a UI harness pushes its operating point off the frontier without exception.

Figure 9 decomposes the gap. Tokens per turn are within 5% across harnesses (~28k–30k), so the cost gap is almost entirely a turn-count gap: 24 turns per coding trial against 171 for computer-use and 219 for browser. The structural reason is action batching. A coding agent writes one script and batches many ERP changes through the JSON-2 API in a single tool call. A UI agent has to wait for the next screen after each form submit, and the per-call action cap (7 for browser, 16 for computer-use) puts a hard ceiling on how many ERP effects fit in one turn. Coding is a natural plan-execute substrate; the UI harnesses are not.

G.5 Reliability Across Repeated Trials

Pass@5 measures whether at least one of five trials succeeds; pass^5 measures whether all five do. Aggregate pass^5 is 9.8% in the coding harness and 3.6% in the browser harness, roughly an order of magnitude below the corresponding pass@5. Frontier agents on ERP-Bench are not yet reliable enough for unattended deployment.

Table 12: Evaluation metrics by model and harness. *Clean* is the share of trials whose terminal state passes every applicable constraint check, *perfect* is the share with reward 100, and *opt|clean* is the mean optimality score among constraint-clean trials. The two halted computer-use runs produced zero points before being stopped.

Harness	Model	Attempts	Parseable	Pass@1	Pass@5	Avg score	Clean	Perfect	Opt|Clean
Coding	GPT-5.5	1,500	1,455	43.4	73.0	63.6	61.0	37.6	90.8
Coding	Claude Opus 4.7	1,500	1,475	30.8	60.7	52.2	42.5	27.6	94.7
Coding	GLM-5.1	1,500	1,476	35.8	63.3	53.7	45.7	30.6	95.6
Coding	Kimi K2.5	1,500	1,410	9.1	20.3	18.5	10.4	8.9	92.8
Browser	GPT-5.5	1,500	1,346	9.7	23.7	16.8	12.7	9.5	90.3
Browser	Claude Opus 4.7	1,500	1,435	28.8	44.3	47.8	40.1	27.3	86.8
Browser	GLM-5.1	1,500	1,439	2.4	12.0	3.5	2.8	2.4	95.1
Browser	Kimi K2.5	1,500	1,452	1.5	7.3	2.7	1.7	1.5	98.5
Computer-use	GPT-5.5	1,500	1,444	7.9	16.7	20.0	9.5	7.7	93.4
Computer-use	Claude Opus 4.7	1,500	1,426	23.7	38.3	40.2	34.0	22.3	88.3
Computer-use	GLM-5V-Turbo	540	449	0.0	0.0	0.0	0.0	0.0	—
Computer-use	Kimi K2.5	619	526	0.0	0.0	0.1	0.0	0.0	—

Table 13: Prevalence of failure families per harness, as percentages of parseable failed trials. Families are not exclusive, so rows do not sum to 100.

Harness	Failed	Demand	Timing	Sourcing	Mfg.	Hygiene
Coding	3,482	13.1	68.9	42.8	35.0	38.0
Browser	5,062	66.0	87.2	89.1	19.4	78.5
Computer-use	3,395	47.4	90.5	85.8	21.1	69.1

H Failure Analysis

Parseable failures fall into five broad families. *Demand and sales* failures miss, mis-price, or fail to confirm the required sales orders. *Timing* failures schedule purchases, manufacturing orders, or deliveries after the customer deadline. *Sourcing* failures violate vendor price, lead time, minimum or maximum quantity, consolidation, screened-vendor, spend-floor, or margin policy. *Manufacturing* failures build infeasible plans with incompatible components, workcenters, or capacities. *Hygiene and state* failures break traceability, confirmation, invoicing, seeded-order handling, repair state, or adjacent-data preservation. The families are not exclusive: one terminal state can break demand, timing, sourcing, and hygiene checks at once. We hold optimality out of this view because a sub-oracle optimality score is only meaningful once the terminal state clears constraints and hygiene.

Table 13 labels each parseable failed trial by the families of failed verifier rules in `rule_results.tsv`. Coding failures are mostly plan-execution failures: the agent has enough information to size demand but schedules production before inputs can legally arrive, mutates confirmed procurements after discovering a planning error, or loses origin links while patching the plan. Browser failures more often start with UI state ambiguity. A browser agent can reason about the right quantities and still leave a sales-order line blank, fail to confirm a quotation, or overwrite a procurement date after recognizing the selected vendor is late. Computer-use failures share the business-rule structure of browser failures, amplified by screenshot-based form editing: the characteristic mistake is committing a partially edited PO or SO because the agent cannot reliably tell whether the cell, datepicker, autocomplete, or source field accepted the intended value.

H.1 Failure Examples

The three examples below are randomly sampled failed trajectories from the released run directories under a fixed seed, filtered to parseable rewards with concrete failed verifier rules. Each example traces the verifier failure to the agent decision that caused it.

Coding: manufacturing before inputs arrive. GLM-5.1 on a medium two-stage build correctly sized demand at 64 finished units, then scheduled manufacturing at the start of the run and forced component receipts through the API before respecting supplier lead times. The verifier flagged `supply_timing_feasible`, `po_delivery_schedule_compliance`, `mo_schedule_compliance`, `component_stock_capacity_compliance`, `mo_component_feasibility`, and `po_origin_traceability`. After the first MOs became invalid, the agent cancelled and recreated them but kept the already-created component POs and tried to repair lineage after the fact, updating `origin` fields to point at the new MOs rather than cancelling and rebuilding the dependent POs.

Browser: proceeding after a broken sales-order row. GPT-5.5 on the same two-stage scenario hit a UI failure while creating a customer sales order—a product autocomplete dropdown closed before the line selection committed—and continued without verifying that the customer, product, quantity, price, commitment date, and confirmation state had persisted. The verifier flagged `demand_coverage`, `deadline_fulfillment`, `sale_revenue`, `supply_timing_feasible`, `po_delivery_schedule_compliance`, `po_price_tier_compliance`, `so_confirmed`, and `po_origin_traceability`. The agent then made a separate sourcing mistake: it identified that a candidate vendor would arrive after the deadline, described the vendor as invalid in its own scratchpad, and still entered the late purchase.

Computer-use: ordering after stock already covers demand. GPT-5.5 on a screened buy-only intake task identified the key fact—only one seeded order should be accepted, and existing finished stock covers it—and then created a purchase order anyway. The verifier flagged `new_spend_margin_policy`, `supply_timing_feasible`, `po_delivery_schedule_compliance`, `po_consolidation_compliance`, `po_min_qty_compliance`, and `po_price_tier_compliance`. The same trace shows the form-entry uncertainty that drove most of those rules: the vendor price field displayed 800.54 instead of the tier price 865.41, and a quantity edit failed to update the line amount. Because no new supply was needed for the accepted demand, any nonzero PO introduced avoidable sourcing, timing, and price-tier obligations.

Table 14: Validity checks against the artifact pairs they constrain, using the notation of fig. 2. A check fires when the two named artifacts disagree on what the task requires.

Check	$I_\theta \leftrightarrow E_\theta$	$I_\theta \leftrightarrow x^*_\theta$	$I_\theta \leftrightarrow V_\theta$	$E_\theta \leftrightarrow x^*_\theta$	$E_\theta \leftrightarrow V_\theta$	$x^*_\theta \leftrightarrow V_\theta$
No-op agent					✓	
Oracle replay				✓		✓
LLM consistency judge	✓	✓	✓	✓	✓	✓
Reward-hacking canary			✓			
Expert spot checks	✓	✓	✓	✓	✓	✓

I Validity Checks

Single-source generation does not eliminate every construction error. The CP-SAT program can encode incomplete business logic, and a renderer can mistranslate a correct specification into one of the artifacts. Five end-to-end checks target each of the four artifact-drift failure modes (table 14). Because the generator is a single program rather than a corpus of hand-authored tasks, any defect a check surfaces is traced to the specification or one of its translation layers, fixed once, and propagated deterministically to every affected instance.

No-op agent. A no-op agent should score zero on every task; a nonzero score means the seeded environment already satisfies the verifier. The no-op agent scores zero on 300 of 300 tasks.

Oracle replay. Replaying the solver's certified plan into the seeded environment should receive full reward; a lower score means setup, oracle, and verifier disagree on the same plan. Oracle replay scores full reward on 300 of 300 tasks.

LLM judge. An LLM judge cross-reads the instruction, environment configuration, oracle, and verifier against the CP-SAT program and flags any artifact that disagrees with the others. The judge reviewed all 300 tasks across 12 iterations during pipeline development. Defects it surfaced and we then fixed at the generator level include the setup script writing the wrong initial on-hand stock for some procurement scenarios, the verifier missing record-lineage checks linking purchase and manufacturing orders to their originating sales orders, and the verifier missing checks for policy clauses that the instruction generator had introduced into the prompt.

Reward-hacking canary. The canary flags any rollout whose terminal state both passes every applicable constraint check and reports an objective strictly better than the certified optimum, which would indicate either a verifier hole or a CP-SAT optimality gap. Across the 16,159 completed trials, zero rollouts triggered the canary.

Expert spot checks. Two domain experts independently completed a stratified sample of 15 tasks by hand in the standard Odoo web client, reading only the instruction. The verifier scored their terminal states at a mean reward of 90.48 across the 30 expert-task trials. The experts hit a broken explicit constraint in 3 of 30 trials, in line with the constraint-clean rate of strong models. Mean expert completion time was 55 minutes per task, comparable to the 60-minute trial budget the agents receive.

J Release

We release the repository containing the task generator and the ERP-Bench dataset in harbor format at erpbench.ai.