PRAX: PROGRAMMABLE SYNTHESIS OF VERIFIABLE ENVIRONMENTS FOR WORKING AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Training capable agents for real-world work requires large collections of diverse, executable, and verifiable environments, yet such environments remain expensive to author and difficult to synthesize reliably. Unlike text-only data, a work environment forms a self-contained system: task instructions, underlying state, tool interactions, and verification logic must remain mutually consistent. Directly generating these components with LLMs often produces environments that appear plausible but are actually unsolvable, inconsistent, or incorrectly scored. We introduce **Prax**, a framework for **programmable synthesis of verifiable work environments**. Prax represents an environment as a structured specification that separates invariant task logic from generative parameters, executable ground-truth derivations, rendering rules, and reference instances. This representation enables scaling along two complementary axes: ***(i)*** instantiating the same task structure across new scenarios, entities, and states, and ***(ii)*** applying correctness-constrained transformations that systematically alter reasoning structure, difficulty, and failure modes. To preserve verifiability, Prax generates environment facts before deriving expected outcomes, and validates each synthesized instance with programmatic checks for structural validity, tool-call closure, state–verifier consistency, answer leakage, and runtime compatibility. We further execute synthesized environments with agents and reject trajectories that fail execution or quality criteria before training. Starting from over 20k synthesized task intents, Prax constructs 25k reusable specifications and over 65k environment instances spanning diverse workflows. Training on the verified trajectories yields state-of-the-art performance on general knowledge tasks at 35B size, demonstrating that Prax can turn a small collection of expert-designed tasks into scalable, high-quality supervision for agents that perform real-world professional work.

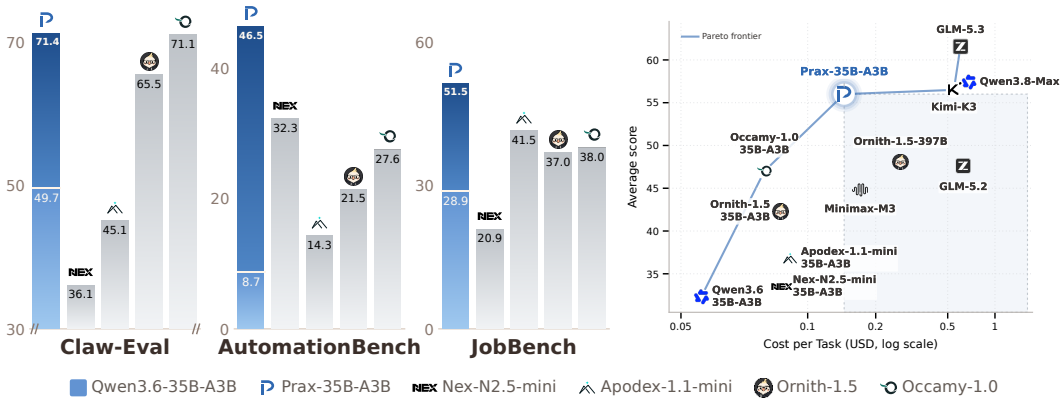


Figure 1: **Benchmark results and cost–performance trade-offs of Prax. Left:** Prax consistently improves upon the Qwen3.6-35B-A3B base model across Claw-Eval, AutomationBench, and JobBench. **Right:** Average benchmark performance versus cost per task.

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.

1 INTRODUCTION

Large language models (LLMs) have evolved from conversational assistants into agentic systems capable of long-horizon planning and tool use (Anthropic, 2026; Meta, 2026). Building on capabilities honed through coding, these agents can now carry out increasingly complex, multi-step workflows with limited supervision (Jimenez et al., 2024; Guo et al., 2025). These capabilities are enabling agents to expand beyond code into general *knowledge work*—reconciling scattered records and coordinating actions across enterprise systems (Shepard & Salimans, 2026b). Unlike coding, where compilers and test suites provide concrete checks, knowledge work often lacks readily available ways to verify whether the work has been completed successfully. A central challenge is thus how to turn knowledge work into a broad supply of diverse training tasks with trustworthy verifiers.

The key challenge in synthesizing realistic working environments is to preserve a reliable correctness. Real-world work often involves stale records, conflicting sources, implicit policies, and irrelevant information, where resolving ambiguity is often part of the task itself rather than simply removing these noisy. Moreover, one request may admit multiple valid actions and outputs, while an apparently correct solution may still violate requirements or rely on prohibited evidence. This makes post-hoc verification difficult: checking only a few observable outcomes may miss important errors, whereas enforcing a specific reference may reject alternative valid solutions. More fundamentally, once the task and environment have been generated, the verifier must infer which evidence matters, how conflicts should be resolved, and which alternative solutions should still be accepted.

For this core challenge, our key idea is to ***co-compile the task, environment, and verifier from a shared executable specification***. Rather than asking a verifier to reconstruct correctness *after* generation, we preserve the task's internal logic that determines what evidence matters, how it should be resolved, and what effects should result. The environment and verification criteria are then derived from this identical logic, keeping what the agent observes and what the verifier rewards consistent. Therefore, verification becomes a product of synthesis, rather than a separate inference problem.

In this paper, we introduce **Prax**, a programmable synthesis framework built around a typed decision language for professional working environment. A Prax program captures the abstract structure of a task: **(*i*)** how information is retrieved and adjudicated, **(*ii*)** how decisions depend on one another, and **(*iii*)** what effects they ultimately produce, while leaving concrete entities and values unspecified. Specifically, the compiler first executes the program in a *demand mode* to construct a world containing the required evidence, conflicts, and distractors under explicit consistency constraints. It then executes the same program in a *solve mode* to derive the expected effects, verifier assertions, a reference trajectory, and targeted incorrect alternatives. Additional static checks and execution-based replay reject instances whose intended decisions are unsatisfiable, unobservable, or incorrectly scored. Every correctness-relevant artifact is thus derived from the same executable source of truth.

This abstraction provides two complementary axes for scaling: (I) *Instance Scaling*. A single decision program can be instantiated across different entities, workspace states, and surface realizations, producing many concrete tasks without changing their underlying reasoning structure. (II) *Structural Scaling*. Programs themselves can also be varied to change information dependencies, decision composition, difficulty, and targeted failure modes. Prax therefore scales not only the number of environments, but the distribution of reasoning structures that working agents can exercise. We further formalize why retaining the executable specification is important. If correctness-relevant information is lost during task generation, a verifier built only from the rendered task may be unable to recover the intended correctness criteria. By co-compiling the task and verifier from the retained specification, Prax eliminates this source of verification ambiguity under the stated assumptions.

Starting from over 20k synthesized task intents, we leverage Prax to produce **25k reusable specifications** and more than **65k executable environment instances**. Training on trajectories from these environments yields substantial improvements across diverse benchmarks: for a 35B-A3B model, Claw-Eval Pass[3] improves from 49.7 to 71.4, AutomationBench pass rate from 8.7 to 46.5, and JobBench score from 28.9 to 51.5. Notably, Prax-35B-A3B achieves competitive performance, even outperforming DeepSeek-v4-Flash-0731 and approaching Kimi-K3. Surprisingly, Prax-4B outperforms all comparable models and achieves AutomationBench pass rate 35.5, where the other small models obtain near 0 scores. Together, these results establish Prax as a scalable and effective framework for constructing verifiable working environments and training capable agents.

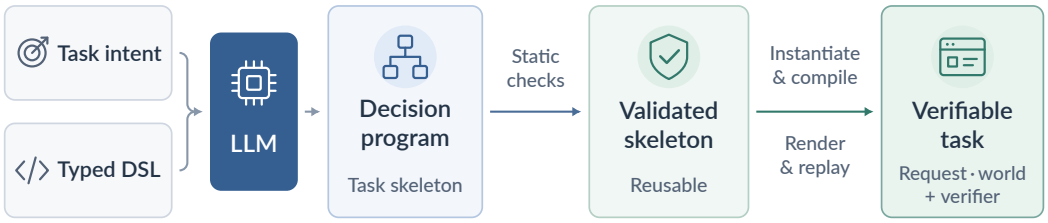


Figure 2: Overview of the Prax environment synthesis framework.

2 PRAX: CO-COMPILING TASKS FROM DECISION PROGRAMS

2.1 PROBLEM FORMULATION

Tasks and verifiers. Let $\mathcal{X}$ denote the space of professional working problems. A problem $x = (m, W_0) \in \mathcal{X}$ comprises a natural-language request m and an initial environment state W_0 . Let $\mathcal{Y}x$ denote the space of a possible rollout trajectory τ for task x, and define $\mathcal{Y} = \bigcup_{x\in\mathcal{X}} \mathcal{Y}_x$. A *verifier* $V = (\mathcal{R}_V, g_V)$ consists of a finite collection of grading rubrics $\mathcal{R}_V$ and an executable scoring function g_V that evaluates a trajectory against those rubrics:

$$g_V(\cdot \mid \mathcal{R}_V) : \mathcal{Y}_x \longrightarrow [0, 1], \qquad A_V = \{\tau \in \mathcal{Y}_x : g_V(\tau \mid \mathcal{R}_V) = 1\}. \tag{1}$$

Thus, we define (x, V) as a *verifiable task*, with A_V denoting its acceptance set and the score 1 indicating full credit. The scoring processing will use any recorded evidence available to the verifier, including outputs, the terminal state W_y, and interaction logs. During verifier construction, we odo not assume a fixed rubric format or partial-credit scheme. We only consider whether a trajectory receives full credit and disregard partial-credit scores.

Task synthesis. Let Ω denote the space of initial task seeds, and let μ be a distribution over Ω. Each seed $w \in \Omega$ specifies the task logic and the concrete information required for task instantiation. Let $\mathcal{V}$ denote the space of verifiers. We define a *task synthesizer* as a map, *i.e.*

$$\mathsf{Syn} : \Omega \longrightarrow \mathcal{X} \times \mathcal{V} \times \mathcal{Y}, \qquad \mathsf{Syn}(w) = (\pi(w), C(w), s(w)) = (x, V, \tau^*), \qquad w \sim \mu. \tag{2}$$

where x is the synthesized task, V is its verifier, and τ^* is the reference trajectory. And π is the problem generator, C is the verifier constructor, and s is the reference generator.

In this construction process, all output must be compatible: a verifier $V = C(w)$ grades trajectories in $\mathcal{Y}_{\pi(w)}$, and the reference trajectory $\tau^* = s(w) \in \mathcal{Y}_{\pi(w)}$. For each seed w, the intended task specification defines a set of correct trajectories $A^*(w) \subseteq \mathcal{Y}_{\pi(w)}$. A successful synthesis should satisfy $\tau^* \in A^*(w)$ and $A_{C(w)} = A^*(w)$, ensuring that the reference trajectory is valid and that the verifier accepts exactly the intended correct solutions.

Verification errors. For a problem x, let μ_x be the conditional distribution of $w \sim \mu$, given $\pi(w) = x$, and let ρ_x be a candidate distribution on $\mathcal{Y}_x$. Drawing $w \sim \mu_x$ and $\tau \sim \rho_x$ independently conditional on x, this models agent behavior that depends on the specific problem. We define false acceptance (FA) of the invalid trajectories and false rejection (FR) of valid trajectories for the verifier constructor C by

$$\begin{aligned} \mathrm{FA}(C \mid x) &= \Pr_{w\sim\mu_x,\, \tau\sim\rho_x} \left[\tau \in A_{C(w)} \setminus A^*(w)\right], \\ \mathrm{FR}(C \mid x) &= \Pr_{w\sim\mu_x,\, \tau\sim\rho_x} \left[\tau \in A^*(w) \setminus A_{C(w)}\right]. \end{aligned} \tag{3}$$

Their sum captures the constructed verifier's disagreement with the intended correctness criterion. Exact verification achieves $A_{C(w)} = A^*(w)$, thereby reducing both false acceptance and false rejection to zero while still permitting multiple valid solutions.

Information available during synthesis. A ***post-hoc*** verifier constructor has access only to the rendered task x rather than the underlying seed w. Formally, it takes the form $C = C_0 \circ \pi$ for some $C_0 : \mathcal{X} \to \mathcal{V}$, where C_0 constructs a verifier from the rendered task alone. If two distinct synthesis inputs w_1 and w_2 produce the same task x but induce different correctness sets, i.e., $\pi(w_1) = \pi(w_2)$ while $A^*(w_1) \neq A^*(w_2)$, then a post-hoc constructor cannot distinguish between them and must recover the missing correctness criterion from x alone. In contrast, a ***co-compiling*** constructor retains access to w and can directly use the underlying task specification when constructing the

verifier. Theorem 1 establishes a lower bound on the verification error of post-hoc constructors, while Theorem 2 gives sufficient conditions under which Prax achieves exact verification.

Instantiation in Prax. Prax constructs w through a structural intent, a typed decision program P, and a world-state assignment, together with the compiler configuration and realized rendering choices (Figure 2). The intent controls workflow coverage (§2.2); the program specifies dependencies, adjudication rules, and required outcomes (§2.3); and the assignment supplies instance values. The generators π, C, and s are implemented as compiler passes over this shared specification (§2.4). One program admits scalable assignments and therefore defines a reusable task family.

2.2 From skill coverage to structural intents

An intent specifies a workflow before its narrative is written. We organize capabilities as a directed bipartite graph $G = (\mathcal{A} \cup \mathcal{S}, E)$, with an artifact $a \in \mathcal{A}$ and a skill $s \in \mathcal{S}$; the edges indicate consumption or production. An alternating artifact–skill path defines a candidate workflow (Figure 3). We constrain path construction to ensure workflow validity: source artifacts can only serve as inputs, generic artifacts such as "document" can only serve as final outputs, and intermediate artifacts must be semantically compatible with adjacent skills. These constraints prevent syntactically valid but semantically implausible workflow compositions. Appendix B.1 provides further details.

We control intent diversity at both structural and surface levels. Valid workflows are sampled by occupation and depth, with quotas and fingerprints limiting imbalance and repetition. Surface properties, such as material condition, user expertise, language, and request specificity, are sampled separately, with incompatible combinations filtered. The resulting intent is translated into a program specification containing service bindings, required primitive families, and a decision-point budget. This connects graph-level coverage to compiler checks: an intent counts toward its quota only after an instance passes acceptance. The user-facing request describes the starting situation and desired result without enumerating the hidden capability path. Intents whose success conditions cannot be expressed by the supported observable contracts are excluded from this synthesis setting.

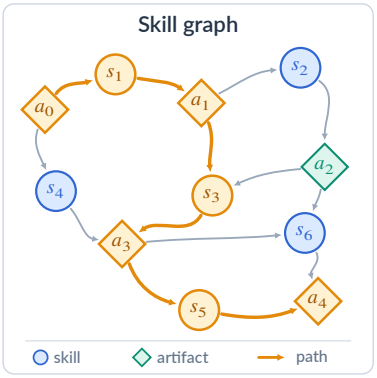


Figure 3: A sampled skill–artifact path.

2.3 Decision programs and primitive contracts

Prax provides a typed builder language implemented in Python for specifying task dependencies and operations. Instance-level attributes, such as entity names, identifiers, dates, and numerical values, are instantiated by the compiler, while schema definitions, operator types, and fixed action labels remain part of the program specification. This separation allows a program to define the required relational and decision structure without committing to particular surface-level instances.

The language of Prax has 4 primitive families. *Source* primitives retrieve entities, policy rules, and referenced records. *Adjudication* primitives resolve recency, authority, ties, or eligibility over finite candidate sets. *Computation* primitives perform arithmetic, aggregation, filtering, and formatting. *Control and sink* primitives branch, iterate over bounded collections, and declare required effects. All primitives and their rules are detailed in Appendix. B.2. Recursion and unbounded loops of the program are excluded, and selection rules must resolve ties explicitly.

Each primitive defines a typed interface together with its execution semantics, materialization requirements, known incorrect alternatives, and tool mapping. Sink primitives further specify service-specific assertions for validating the resulting state changes. For example, `resolve_latest` operates over timestamped candidates, selects the latest applicable value, and retains an older candidate as a counterfactual. We perform static checks for type consistency, tool and assertion support, primitive coverage, decision budget, and connectivity from task-relevant source to downstream actions. Because structural connectivity alone does not guarantee behavioral relevance, instance-level con-

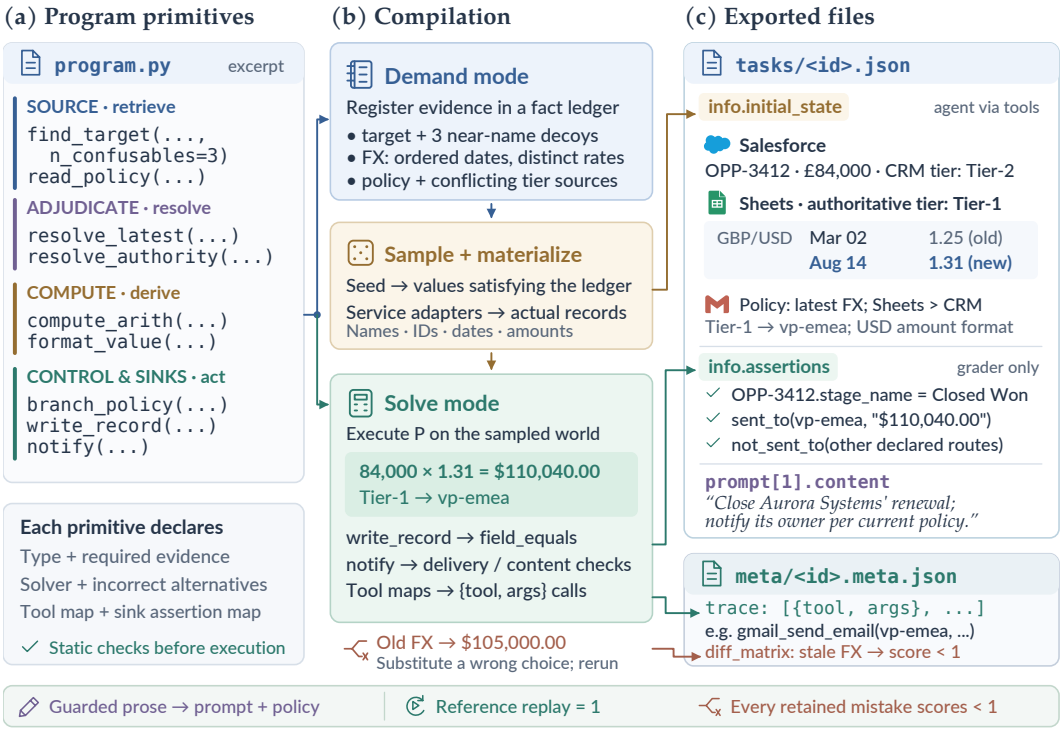


Figure 4: **From primitive contracts to runnable task files.** Demand mode collects evidence requirements; sampling and service adapters materialize the workspace. Solve mode exports assertions and tool calls using the same values.

straints and counterfactual differencing are used to verify that changing a decision-relevant value can affect the final outcome. Invalid programs are repaired when possible and otherwise discarded.

2.4 Compiling the World and Its Verifier

Demand execution and constrained instantiation. In demand mode, primitive calls register requirements instead of returning concrete values. The resulting *fact ledger* records entities, references, candidate sets, policy rules, and constraints. For Figure 4, it requires a uniquely identifiable opportunity among decoys, an ordered pair of FX rows, conflicting tier values with a specified authoritative source, and a routing policy. Constraints also prevent ineffective traps: the older rate must change the rendered amount, and the stale tier must select a different route.

The sampler assigns names, identifiers, timestamps, and numeric values subject to this ledger. Service adapters place the assigned facts in structured records and language carriers. Selection criteria, source precedence, and policy applicability are materialized alongside the evidence. Failed sampling attempts are retried within a budget; contradictory specifications are returned for repair or discarded.

Solve execution and artifact export. The compiler executes the same program on the instantiated world to obtain decisions d^* and a reference terminal state W^*. It derives three artifacts:

- **Verifier.** Each sink exports assertions for its declared effects, with arguments taken from the solved values. Notification sinks also export negative assertions over unselected recipients in the declared routing domain, preventing broadcast-to-all strategies within that domain. Content checks follow the sink contract; they need not require an exact copy of the reference message. These Boolean assertions instantiate the rubrics $\mathcal{R}_V$. Prax uses a weighted mean with strictly positive weights, so $g_V(y) = 1$ exactly when all assertions hold (Appendix B.3).
- **Reference trajectory.** Primitive tool mappings are instantiated and composed in dependency order, including bindings to identifiers returned by earlier calls.

- **Counterfactual catalogue.** The compiler substitutes a declared incorrect choice at a decision site and recomputes downstream effects, producing targeted mistakes such as using the stale rate or notifying the wrong team.

For example, an amount of £84,000 and a valid rate of 1.31 produce $110,040; using a stale rate of 1.25 produces $105,000. The same program derives both outcomes and the notification assertion that distinguishes them. Appendix C follows this instance through compilation.

Controlled rendering. Structured values are instantiated directly by code, while natural-language artifacts are rendered with constrained LLM variation around required clauses and placeholders. We check the rendered text for semantic anchors, placeholder integrity, and answer leakage before restoring concrete values. Values or conclusions that the agent must derive are withheld from the request but remain recoverable from workspace evidence. Exactness assumes semantic preservation during rendering, with automated checks covering common violations. Reproduction retains the program, seed, configuration, toolchain versions, and cached renderings.

2.5 Acceptance and difficulty control

Acceptance links the compiled artifacts back to executable behavior. First, the reference trajectory is replayed through the actual service tools and is required to receive full credit. This checks that a valid solution is executable and accepted. Second, each retained counterfactual is replayed from the same clean initial state W_0 and must receive less than full credit.

$$g_V(\tau^*) = 1, \qquad g_V(\tau_j^-) < 1 \quad \text{for every declared mistake } j. \tag{4}$$

The per-assertion failure vector identifies the requirements violated by each counterfactual. These checks establish task solvability and verifier sensitivity to designed errors, though they do not cover all possible agent behaviors. Normalized program fingerprints and batch-level overlap audits control structural duplication and potential evaluation contamination. Failure diagnostics separate program, instantiation, and rendering errors to guide targeted retries. Difficulty controls and calibration are detailed in Appendix B.3.

3 What co-compilation guarantees

We separate information loss from compiler fidelity; proofs and scope details are in Appendix A.

Information available to the verifier. Using the conditional distributions in §2.1, define

$$p_x(\tau) = \Pr_{w \sim \mu_x}[\tau \in A^*(w)], \qquad \mathrm{Amb}(x) = \mathbb{E}_{\tau \sim \rho_x} \min\{p_x(\tau), 1 - p_x(\tau)\}. \tag{5}$$

Here $\mathrm{Amb}(x)$ measures disagreement about correctness among specifications consistent with x.

Theorem 1 (Information limit of post-hoc verification)**.** *Every post-hoc constructor $C = C_0 \circ \pi$ satisfies*

$$\mathrm{FA}(C \mid x) + \mathrm{FR}(C \mid x) \geq \mathrm{Amb}(x). \tag{6}$$

The bound is tight over unrestricted acceptance rules.

This shows that unresolved uncertainty about task correctness imposes an unavoidable error floor on post-hoc verification.

Fidelity to an observable contract. For a fixed Prax instance, let $q_P(W)$ collect sink-declared field values, content features, and prohibited-effect indicators within a bounded domain. With W^* the solve-mode state, define

$$A^*(w) = \{\tau \in \mathcal{Y}_{\pi(w)} : q_P(W_\tau) = q_P(W^*)\}. \tag{7}$$

This admits different trajectories and unconstrained wording with the same required effects.

Theorem 2 (Fidelity of a co-compiled verifier)**.** *For a finite Prax program with a world satisfying its demand ledger, assume (i) sound demand contracts and total, deterministic primitive evaluation, (ii) rendering and tool execution preserve primitive semantics, (iii) exact assertion predicates cover every component of q_P without extra constraints, and (iv) full credit is awarded if and only if all assertions hold. Then $\mathsf{Syn}(w) = (x, V, \tau^*)$ satisfies*

$$g_V(\tau^*) = 1, \qquad A_V = A^*(w). \tag{8}$$

If these assumptions hold for every generated instance, both error probabilities are zero for any candidate distribution.

This shows that the compiled verifier accepts exactly the solutions under the declared task contract.

Table 1: Performance of Prax and baseline models on four working-related benchmarks. **Bold** and underlined values indicate the best and second-best results, respectively.

Model	Claw-Eval			AutomationBench		JobBench		WorkspaceBench				Avg.
	Pass^3	Pass@3	Score	Pass Rate	Score	Easy	Main	Total	Easy	Medium	Hard	
Frontier Models												
GLM-5.3	72.2	89.2	83.8	**48.2**	81.8	65.0	**66.3**	**59.7**	**70.4**	**63.8**	49.5	**61.6**
Qwen3.8-Max	**72.8**	89.1	85.1	41.8	80.1	**78.7**	59.8	55.1	64.3	60.2	43.7	57.4
Kimi-K3	69.3	**93.4**	**85.4**	46.7	80.4	77.5	55.8	54.1	51.3	56.7	**50.9**	56.5
Open source models (> 300B)												
DeepSeek-v4-Flash-0731	**72.8**	87.8	81.4	38.5	76.7	78.5	52.9	49.1	57.8	53.2	39.6	53.3
Ornith-1.5-397B	70.9	86.7	82.0	17.7	46.6	71.3	48.3	55.7	60.4	59.0	48.7	48.1
GLM-5.2	67.9	83.6	78.0	20.3	51.7	77.5	48.1	54.0	63.9	53.7	50.8	47.6
Minimax-M3	66.3	85.8	81.3	19.3	57.7	72.4	41.3	52.4	47.4	56.2	48.4	44.8
Open source models (35B-A3B)												
Occamy-1.0 35B-A3B	71.1	84.9	80.6	27.6	69.3	**76.5**	38.0	51.1	42.6	54.9	**48.4**	47.0
Ornith-1.5-35B-A3B	65.5	**87.2**	79.0	21.5	59.7	69.3	37.0	45.3	52.6	47.1	39.9	42.3
Apodex-1.1-mini 35B-A3B	45.1	72.6	65.8	14.3	51.5	48.3	41.5	46.2	54.3	50.2	36.9	36.8
Qwen3.6-35B-A3B	49.7	72.5	69.5	8.7	40.7	28.9	28.9	41.9	41.7	44.5	38.0	32.3
Nex-N2.5-mini 35B-A3B	36.1	70.5	65.8	32.3	47.8	61.6	20.9	44.8	43.5	45.7	43.8	33.5
Ours												
Prax-35B-A3B	**71.4**	86.3	**83.1**	**46.5**	**79.4**	63.6	**51.5**	**54.5**	**57.8**	**59.9**	44.8	**56.0**
Prax-4B	65.5	81.0	79.1	35.5	62.3	35.9	5.1	24.2	49.9	19.7	20.5	32.6

4 EXPERIMENTS

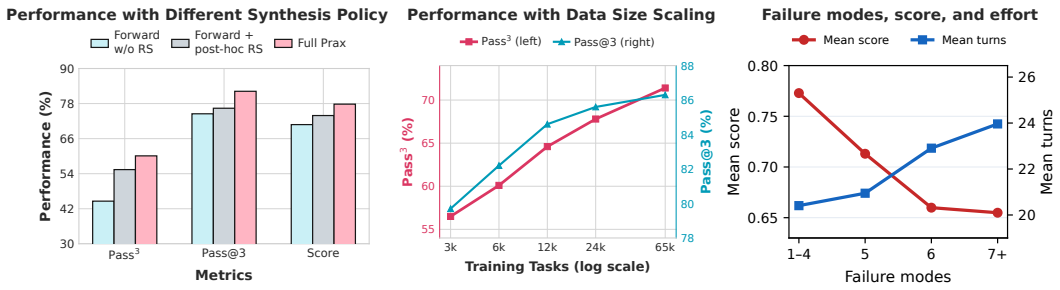
4.1 EXPERIMENTAL SETUP

Benchmarks. We evaluate models across four benchmarks: **(1) Claw-Eval** (Ye et al., 2026), where we use the General subset, report Pass^3, Pass@3, and the average score, and use Gemini-3.5-Flash as the LLM judge; **(2) AutomationBench** (Shepard & Salimans, 2026a) (v1.0.6), where we report both pass rate and score; **(3) JobBench** (Li et al., 2026b), where we use its Easy and Main test sets; and **(4) WorkspaceBench** (Tang et al., 2026), where we report both the overall score and scores across difficulty levels. Additionally, We report the average score **Avg.** of Claw-Eval Pass^3, AutomationBench Pass Rate, JobBench (Main), and WorkspaceBench (Total). For all benchmarks, we follow the official instructions, workspace settings, hyperparameters, and budgets, while retaining their native context-compaction mechanisms. For JobBench and WorkspaceBench, we use a unified Claude Code harness and Claude-Sonnet-4.6 as the LLM judge.

Baselines. We primarily compare our Prax model against 12 representative models spanning different accessibility levels, model families, and parameter scales, including **(1) Frontier models**: GLM-5.3 (GLM-5-Team, 2026), Qwen3.8-Max (Qwen Team, 2026b), Kimi-K3 (Kimi Team, 2026), and Claude-Opus-4.7; **(2) Open-source models with more than 300B total parameters**: DeepSeek-V4-Flash-0731 (Xu et al., 2026), Ornith-1.5-397B (Ornith Team, 2026), GLM-5.2, and Minimax-M3 (Lai et al., 2026); and **(3) Open-source models at the 35B-A3B scale**: Occamy-1.0 (Occamy Team, 2026), Ornith-1.5-35B-A3B, Apodex-1.1-mini (Apodex Team, 2026), Qwen3.6-35B-A3B (Qwen Team, 2026a), and Nex-N2.5-mini 35B-A3B. Additionally, for the 4B variant of Prax, we compare against its backbone, Qwen3.5-4B, as well as recent compact LLMs of comparable scale, including LFM2.5-2.6B (Liquid AI, 2026), Ling-3.0-Tiny (7.9B-A1.3B), MiniCPM5-2B, and Gemma-4-E4B-it (Gemma Team, 2026). We use official APIs for closed-source models and open-source models larger than 300B, while other models are deployed with vLLM (Kwon et al., 2023) framwork. For models that expose a configurable reasoning-effort option, we use the highest available setting (e.g. *xhigh* and *max*).

4.2 MAIN RESULTS

As shown in Table 1, we report the performance of the evaluated models across four work-related benchmarks. Our Prax-35B-A3B achieves an average score of 56.0, outperforming all evaluated open-source baselines (35BA3B) and exceeding Occamy-1.0 by 9.0 points. It also surpasses DeepSeek-V4-Flash-0731 in average score, while approaching the performance of Kimi-K3 and Qwen3.8-Max. Compared with the Qwen3.6-35B-A3B base model, Prax improves the average



(a) Synthesis policy ablation. (b) Training data scaling. (c) Failure modes and effort.

Figure 5: Synthesis policy, data scaling, and task difficulty. (a) Performance with different synthesis policy. (b) Performance with data size scaling. (c) Mean score and assistant turns distributions.

score by 23.7 (+73.4%). The gains are consistent across all four headline metrics: Claw-Eval pass^3 increases from 49.7 to 71.4 (+43.7%), AutomationBench pass rate from 8.7 to 46.5 (+434.5%), JobBench Main from 28.9 to 51.5 (+78.2%), and WorkspaceBench Total from 41.9 to 54.5 (+30.1%). These consistent gains across diverse benchmarks suggest that training on Prax-synthesized environments transfers beyond the synthetic training distribution and improves reusable capabilities for long-horizon tool use and workflow execution.

Results on compact models. The improvement also extends to the smaller model setting. Our Prax-4B obtains a Claw-Eval pass^3 of 65.5 and an AutomationBench pass rate of 35.5. Notably, while all other compact baselines achieve near 0 pass rates on AutomationBench, Prax-4B remains substantially effective and even exceeds all evaluated 35B-A3B baselines. We further compare our Prax-4B with compact open-source baselines, as shown in Table 2. Prax-4B achieves the strongest performance across all metrics, with an average score of 50.5, compared with 22.7 for its Qwen3.5-4B backbone (+122.5%). In particular, it improves Claw-Eval Pass^3 from 43.5 to 65.5 (+50.6%) and AutomationBench Pass Rate from 1.8 to 35.5. The substantial gain on AutomationBench highlights the effectiveness of training on environments synthesized by Prax in improving tool-use capabilities for realistic work environments, even at the 4B scale.

Table 2: Performance of smaller models on Claw-Eval and AutomationBench. **Bold** and underlined values indicate the best and second-best results, respectively.

Model	Claw-Eval			AutomationBench		Avg.
	Pass^3	Pass@3	Score	Pass Rate	Score	
Qwen3.5-4B	43.5	69.1	66.0	1.8	16.9	22.7
LFM2.5-2.6B	36.4	57.3	63.7	0.0	0.9	18.2
Ling-3.0-Tiny	34.7	57.5	62.5	1.0	6.1	17.9
MiniCPM5-2B	32.4	47.2	60.4	0.5	3.1	16.5
Gemma-4-E4B-it	17.2	23.1	45.9	0.0	1.5	8.6
Prax-4B (***Ours***)	**65.5**	**81.0**	**79.1**	**35.5**	**62.3**	**50.5**

Cost–Performance Trade-off. We analyze the cost–performance trade-off across four benchmarks, with per-task costs estimated from model input and output token usage based on the official API pricing of each model. Detailed cost for each model are provided in Appendix F. As shown in Figure 1 (Right), our Prax-35B-A3B lies on the Pareto frontier and achieves a favorable balance between performance and inference cost. At an estimated cost of $0.142 per task, Prax-35B-A3B attains an average score of 56.0,. Moreover, Prax-35B-A3B approaches Kimi-K3 within 1.5 points while reducing the estimated per-task cost by approximately 74%, and remains substantially less expensive than other frontier-scale models. These results demonstrate that Prax delivers strong agentic performance with a competitive inference-cost profile.

4.3 Dataset Analysis

Diversity. As shown in Figure 6 in Appendix D, we analyze tool-use coverage over 64,508 training records. We examine the frequencies of invoked tool families and capability categories to characterize the behavioral diversity in the training data. The resulting distribution spans a broad range of tool families and functional capability domains, indicating that the training mixture covers heterogeneous modes of interaction rather than concentrating on a small set of operations.

Ablation study of synthesis policy. We use 6,000 training tasks for each of three methods: forward synthesis without verifier-based rejection sampling (RS), forward synthesis with a post-hoc verifier followed by RS, and Full Prax. As shown in Figure 5a, we compare their Claw-Eval performance in all metrics. Adding a post-hoc verifier and RS improves Pass^3 from 44.6 to 55.4 (+24.2%), but both baselines remain below Prax at 60.1 (**+34.8%**). Prax also achieves the highest Pass@3 and average score, at 82.2 and 77.8, respectively. These results show that Prax provides more effective training supervision than either forward-synthesis baseline at the same size of tasks.

Difficulty. To examine whether richer failure-mode design corresponds to greater task difficulty, we analyze 5,067 recorded rollouts over synthesized task skeletons, using API tools and a 50-turn budget. For each skeleton, we count the number of declared failure modes and group tasks into different number of modes. Each task contributes one saved rollout, including unsuccessful attempts, while later rejection-sampling rounds are excluded. This count reflects the number of annotated failure modes. Figure 5c shows that the mean score of synthesized tasks decreases from 77.3 to 65.5, while mean assistant turns increase from 20.4 to 24.0. This suggests that tasks with more declared failure modes tend to require longer interactions and yield lower completion quality. Additionally, we aslo provide Contamination analysis in Appendix E.

5 Related Work

Verifiable Working Agent Environments. Recent advances in LLM-based agents have shifted evaluation from static question answering toward goal-directed interaction with executable environments, such as τ-Bench (Yao et al., 2025) and τ^2-Bench (Barres et al., 2026), AgentBench (Liu et al., 2024) and WorkArena (Drouin et al., 2024). More recent benchmarks increasingly target realistic work settings, including GDPVal (Patwardhan et al., 2026), Claw-Eval (Ye et al., 2026), AutomationBench (Shepard & Salimans, 2026a), JobBench (Li et al., 2026b), and WorkspaceBench (Tang et al., 2026). Together, these benchmarks move agent evaluation toward consequential multi-step daily and professional work rather. However, constructing such expert working environments typically requires substantial manual effort to define tasks, initialize states, configure tools, and implement task-specific verification, making executable environments difficult to scale.

Environment and Task Synthesis. EnvScaler (Song et al., 2026) synthesizes environment skeletons, task scenarios, and rule-based validation functions. Agent World Model (Wang et al., 2026) generates executable environments backed by databases for agent training. Related work also explores multi-turn data generation with APIGen-MT (Prabhakar et al., 2025), procedural environment generation (Sullivan et al., 2025), Claw-Gym (Bai et al., 2026), ClawEnvKit (Li et al., 2026a) and trajectory-based environment reconstruction (Wu et al., 2026), yet these approaches primarily target general-purpose tool-interaction and emphasize scaling the diversity of sandboxes, tools, and task scenarios. Prax focuses on environment synthesis specifically for ***professional work scenarios***, where tasks are grounded in realistic workflows, operate over persistent states, and are expected to produce externally verifiable artifacts.

6 Conclusion

We introduced **Prax**, a framework for synthesizing verifiable environments for training LLM agents in professional work scenarios. Prax represents task logic with typed decision programs, enabling environments to scale both through diverse instantiations and through variations in reasoning structure. By retaining executable specifications and co-compiling tasks with their verifiers, Prax preserves task correctness without constraining agents to a single reference trajectory. Experiments across multiple work-oriented agent benchmarks show that training on Prax-synthesized environments consistently improves both compact and larger models, with gains transferring beyond the synthetic training distribution. These results suggest that scalable, verifier-grounded environment synthesis can provide an effective foundation for training more capable agents for realistic work.

AI use statement

Generative AI tools were used to assist with language polishing, translation, and code refinement and debugging. All research ideas, methodological choices, experimental settings, analyses, and

conclusions were developed and verified by the authors. We manually reviewed all AI-assisted outputs for correctness and take full responsibility for the final content.

REFERENCES

Anthropic. Anthropic economic index report: Cadences. Anthropic Research, June 2026. URL https://www.anthropic.com/research/economic-index-june-2026-report.

Apodex Team. Apodex 1.1: Scaling agentic intelligence for complex work. 2026.

Fei Bai, Huatong Song, Shuang Sun, Daixuan Cheng, Yike Yang, Chuan Hao, Renyuan Li, Feng Chang, Yuan Wei, Ran Tao, et al. Clawgym: A scalable framework for building effective claw agents. *arXiv preprint arXiv:2604.26904*, 2026.

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik R Narasimhan. τ^2-bench: Evaluating conversational agents in a dual-control environment. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=OC2z7iSQKa.

Alexandre Drouin, Maxime Gasse, Massimo Caccia, Issam H. Laradji, Manuel Del Verme, Tom Marty, David Vazquez, Nicolas Chapados, and Alexandre Lacoste. Workarena: how capable are web agents at solving common knowledge work tasks? In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

Gemma Team. Gemma 4 technical report, 2026. URL https://arxiv.org/abs/2607.02770.

GLM-5-Team. Glm-5: from vibe coding to agentic engineering, 2026. URL https://arxiv.org/abs/2602.15763.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, et al. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature*, 645:633–638, 2025. doi: 10.1038/s41586-025-09422-z.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Kimi Team. Kimi k3: Open frontier intelligence. 2026.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

Xunhao Lai, Weiqi Xu, Yu-Feng Yang, Qiaorui Chen, Yangyifan Xu, Lunbin Zeng, Xiaolong Li, Haohai Sun, Haichao Zhu, Vito Zhang, and Pengyu Zhao. Minimax sparse attention. *ArXiv*, abs/2606.13392, 2026.

Xirui Li, Ming Li, Ion Stoica, Cho-Jui Hsieh, and Tianyi Zhou. Clawenvkit: Automatic environment generation for claw-like agents. *arXiv preprint arXiv:2604.18543*, 2026a.

Yuetai Li, Yichen Feng, Zhangchen Xu, Zixian Ma, et al. Jobbench: Aligning agent work with human will, 2026b. URL https://arxiv.org/abs/2605.26329.

Liquid AI. Lfm2.5-2.6b: Agents everywhere. *Liquid AI Blog*, 2026. www.liquid.ai/blog/lfm2-5-2-6b.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. Agentbench: Evaluating LLMs as agents. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=zAdUB0aCTQ.

Meta. Introducing muse. Meta AI, 2026. URL https://ai.meta.com/muse/.

Occamy Team. Occamy-1.0: Open pareto-frontier 35b intelligence for co-work. 2026.

Ornith Team. Ornith-1.5: From self-scaffolding to self-improvement, 2026. URL https://ornith.ai/ornith_1_5.html.

Tejal Patwardhan, Rachel Dias, Elizabeth Proehl, Grace Kim, Michele Wang, Olivia Watkins, Simon Fishman, Marwan Aljubeh, Phoebe Thacker, Laurance Fauconnet, Natalie Kim, Samuel Miserendino, Gildas Chabot, David Li, Patrick Chao, Michael Sharman, Alexandra Barr, Amelia Glaese, and Jerry Tworek. Gdpval: Evaluating ai model performance on real-world economically valuable tasks. In C. Vondrick, B. Hariharan, C. Raffel, L. Pinto, D. Yang, and A. Faust (eds.), *International Conference on Learning Representations*, volume 2026, pp. 24005–24040, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/290c2430f91912204f30bbcc990fff1d-Paper-Conference.pdf.

Akshara Prabhakar, Zuxin Liu, Ming Zhu, Jianguo Zhang, Tulika Manoj Awalgaonkar, Shiyu Wang, Zhiwei Liu, Haolin Chen, Thai Hoang, Juan Carlos Niebles, Shelby Heinecke, Weiran Yao, Huan Wang, Silvio Savarese, and Caiming Xiong. Apigen-mt: Agentic pipeline for multi-turn data generation via simulated agent-human interplay. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference. Curran Associates, Inc., 2025. doi: 10.52202/085713-2184. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/5e3661f7fe4c8ac5652d62eb3d3c96ea-Paper-Datasets_and_Benchmarks_Track.pdf.

Qwen Team. Qwen3.6-35B-A3B: Agentic coding power, now open to all, April 2026a. URL https://qwen.ai/blog?id=qwen3.6-35b-a3b.

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

Dan Shepard and Robin Salimans. Automationbench. *ArXiv*, abs/2604.18934, 2026a.

Daniel Shepard and Robin Salimans. AutomationBench, 2026b. 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. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 8326–8357, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-395-1. doi: 10.18653/v1/2026.findings-acl.407. URL https://aclanthology.org/2026.findings-acl.407/.

Michael Sullivan, Mareike Hartmann, and Alexander Koller. Procedural environment generation for tool-use agents. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 18544–18562, Suzhou, China, November 2025. Association for Computational Linguistics. ISBN 979-8-89176-332-6. doi: 10.18653/v1/2025.emnlp-main.936. URL https://aclanthology.org/2025.emnlp-main.936/.

Zirui Tang, Xuanhe Zhou, Yumou Liu, Linchun Li, Weizheng Wang, Hongzhang Huang, Jun Zhou, Jiachen Song, Shaoli Yu, Jinqi Wang, Zihang Zhou, Hongyi Zhou, Yuting Lv, Jinyang Li, Jiashuo Liu, Ruoyu Chen, Chunwei Liu, GuoLiang Li, Jihua Kang, and Fan Wu. Workspace-bench 1.0: Benchmarking ai agents on workspace tasks with large-scale file dependencies, 2026. URL https://arxiv.org/abs/2605.03596.

Zhaoyang Wang, Canwen Xu, Boyi Liu, Yite Wang, Siwei Han, Zhewei Yao, Huaxiu Yao, and Yuxiong He. Agent world model: Infinity synthetic environments for agentic reinforcement learning. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=OvLnVkV14P.

Jie Wu, Zhenru Zhang, Beichen Zhang, Xuwu Wang, Yuhui Su, Mouxiang Chen, Peng Wang, Zhihai Wang, Que Shen, Hao Zhou, et al. Terminal-universe: Turning agent trajectories into scalable terminal environments. *arXiv preprint arXiv:2609.04148*, 2026.

Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chenchen Ling, et al. Deepseek-v4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348*, 2026.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik R Narasimhan. {τ}-bench: A benchmark for \underline{T}ool-\underline{A}gent-\underline{U}ser interaction in real-world domains. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=roNSXZpUDN.

Bowen Ye, Rang Li, Qibin Yang, Yuanxin Liu, Linli Yao, Hanglong Lv, Zhihui Xie, Chenxin An, Lei Li, Lingpeng Kong, Qi Liu, Zhifang Sui, and Tong Yang. Claw-eval: Towards trustworthy evaluation of autonomous agents, 2026. URL https://arxiv.org/abs/2604.06132.

A Proofs and scope of the guarantees

A.1 Setup and the post-hoc bound

Fix a synthesis input space Ω, seed distribution μ, publication map π, and correctness map A^* as in §2.1. The fiber $\Omega_x = \{w \in \Omega : \pi(w) = x\}$ collects inputs consistent with the publication; μ_x is the conditional seed law on this fiber. We assume measurable spaces and maps admitting these conditional laws, and all conditional statements hold for almost every x under the distribution induced by π and μ. A candidate y is drawn from ρ_x, independently of w conditional on x. This covers a fixed policy interacting with the published environment when its observations depend only on that environment.

Write $b_w(\tau) = \mathbf{1}[\tau \in A^*(w)]$ and $v_w^C(\tau) = \mathbf{1}[\tau \in A_{C(w)}]$. The quantities in Eq. 3 are joint error probabilities, rather than error rates conditioned on the true class:

$$\mathrm{FA}(C \mid x) = \mathbb{E}_{w\sim\mu_x}\mathbb{E}_{\tau\sim\rho_x}\big[v_w^C(\tau)(1 - b_w(\tau))\big], \tag{9}$$

$$\mathrm{FR}(C \mid x) = \mathbb{E}_{w\sim\mu_x}\mathbb{E}_{y\sim\rho_x}\big[(1 - v_w^C(\tau))b_w(\tau)\big]. \tag{10}$$

This definition makes their sum the probability that the verifier disagrees with the specified contract. For randomized verifiers, the expectations also average over verifier randomness. A bound on this sum does not imply a separate positive lower bound on each term.

Lemma 1 (Constancy on a publication fiber)**.** *For a deterministic post-hoc constructor $C_0 : \mathcal{X} \to \mathcal{V}$ and $C = C_0 \circ \pi$, the acceptance decision $v_w^C(\tau)$ for a fixed τ is the same for every $w \in \Omega_x$. For a randomized constructor whose randomness is independent of (w, τ) conditional on x, the acceptance probability has the same property.*

Proof. The constructor C_0 receives the same input x for every member of Ω_x. In the randomized case its conditional input and randomness distributions are likewise identical. □

Proof of Theorem 1. For $C = C_0 \circ \pi$, let $r_x(\tau) \in [0, 1]$ be the probability that $C_0(x)$ accepts τ, including any randomness in construction or evaluation. By Lemma 1, this probability cannot depend on which synthesis input produced x. Conditional on x, τ, the expected disagreement is

$$r_x(\tau)(1 - p_x(\tau)) + (1 - r_x(\tau))p_x(\tau) \;\geq\; \min\{p_x(\tau), 1 - p_x(\tau)\}. \tag{11}$$

The inequality follows because the left side is a convex combination of $1 - p_x(\tau)$ and $p_x(\tau)$. Averaging over $\tau \sim \rho_x$ gives Eq. 6. Equality is attained by $r_x(\tau) = 1$ when $p_x(\tau) > 1/2$ and $r_x(\tau) = 0$ when $p_x(\tau) < 1/2$, with any choice at a tie. This is tightness over unrestricted decision rules; a restricted verifier language need not represent the Bayes rule, and the theorem does not assert that its posterior is computationally accessible. □

An example of information loss. Suppose a generated workspace contains rates 1.25 and 1.31, but its request and policy records omit whether to use the booking-date or settlement-date rate. Consider a controlled generator with two equally likely specifications that produce this identical publication while selecting different dates. For a base amount of $84{,}000$, the two contracts require $105{,}000$ and $110{,}040$, respectively. A candidate distribution supported equally on these two outcomes has $p_x(\tau) = 1/2$ and $\mathrm{Amb}(x) = 1/2$. If the governing date convention is instead published, this source of ambiguity disappears. The example illustrates a defective publication, not a source of desirable task difficulty.

Conservative and permissive acceptance. For a finite posterior support, accepting only the intersection of its specified accept sets gives zero false acceptance but may reject outcomes correct under the sampled specification. Accepting their union gives zero false rejection but may admit incorrect outcomes. These are extremes of the tradeoff; neither generally attains the Bayes bound. In particular, checking fewer necessary conditions enlarges an accept set and can increase false acceptance. It should not be identified with the conservative intersection rule.

What zero ambiguity means. $\mathrm{Amb}(x) = 0$ exactly when $p_x(\tau) \in \{0, 1\}$ for ρ_x-almost every candidate. This is agreement about correctness under the posterior, up to null sets. It permits multiple correct trajectories and does not require a singleton latent specification. Nor does it imply that a computationally limited verifier can recover the contract efficiently. Theorem 1 isolates information loss; it is not a computational lower bound.

A.2 Coarsening the published information

Corollary 1 (Bayes error under information coarsening)**.** *Let $w \sim \mu$, $X = \pi(w)$, and let X' be a coarsened publication such that $w \to X \to X'$ is a Markov chain. Fix a common candidate distribution ρ independent of (w, X, X'), using a shared candidate representation under which the correctness event $\tau \in A^*(w)$ is unchanged. For $Z \in \{X, X'\}$, write $p_Z(\tau) = \Pr[\tau \in A^*(w) \mid Z]$ and $\mathrm{Amb}(Z) = \mathbb{E}_{\tau\sim\rho} \min\{p_Z(\tau), 1 - p_Z(\tau)\}$. Then*

$$\mathbb{E}\big[\mathrm{Amb}(X')\big] \geq \mathbb{E}\big[\mathrm{Amb}(X)\big]. \tag{12}$$

Proof. For each fixed y, conditional independence and the tower property give $p_{X'}(\tau) = \mathbb{E}[p_X(\tau) \mid X']$. The function $h(p) = \min\{p, 1 - p\}$ is concave, so Jensen's inequality yields

$$h(p_{X'}(\tau)) \geq \mathbb{E}[h(p_X(\tau)) \mid X'].$$

Take expectations over X' and then over the common ρ. □

The result concerns expected information loss, not realism or pointwise task difficulty. An invertible re-encoding or the addition of independent distractors while retaining the original information can leave the bound unchanged. Transformations depending on private criteria need not satisfy the Markov condition. If a presentation changes which actions or identifiers a trajectory refers to, a common outcome representation is required before applying the corollary. Changes in a policy's rollout distribution also fall outside the fixed-ρ comparison. Thus neither more distractors nor more decision points implies a strictly larger ambiguity bound.

A.3 Compiler contracts and fidelity

Fix a synthesis input w for Prax and write $\mathsf{Syn}(w) = (x, V, \tau^*)$, so $x = \pi(w)$, $V = C(w)$, and $\tau^* = s(w)$. The specification in Eq. 7 is defined at the level of primitive and sink semantics, independently of the emitted verifier code. Let $a_1, \ldots, a_n$ be the Boolean terminal-state assertions emitted as Prax's grading rubrics. For this program and instance, write the observable projection as $q_P(W) = (q_1(W), \ldots, q_K(W))$. A component may be a record-field value, the presence of required content in a qualifying notification, or an indicator of a prohibited delivery. A content-presence component deliberately permits wording outside the required content to vary. Requiring the absence of a delivery is a statement about the final sent-message record; it does not imply that an unlogged transient action never occurred.

The compiler-correctness assumptions of Theorem 2 can be decomposed into the following obligations.

1. **Demand soundness and total evaluation.** A satisfying instance supplies the required records, closed references, applicable policies, and finite candidate sets. Arithmetic and formatting operations are defined on the sampled inputs. Selection either has a unique winner under the declared rule or a specified tie-break; empty sets follow explicit control branches.
2. **Faithful execution and rendering.** The materialized records and policy carriers preserve those facts and rules. Each primitive's tool-call realization implements its abstract read or effect, including returned-ID bindings. Bounded compositions preserve these semantics.
3. **Exact and complete assertion export.** Every declared component has an exact predicate, all required positive and negative obligations are exported, and no extra grading obligations are introduced. If I_k indexes the assertions for component k, then for every reachable terminal state,
$$\bigwedge_{i \in I_k} a_i(W) \quad \Longleftrightarrow \quad q_k(W) = q_k(W^*). \tag{13}$$
Every assertion belongs to a declared component.
4. **Full-credit scoring.** For every feasible trajectory y, the scoring rule satisfies
$$g_V(\tau) = 1 \quad \Longleftrightarrow \quad \bigwedge_{i=1}^{n} a_i(W_\tau). \tag{14}$$
Prax's positive weighted mean satisfies this condition, as does any other aggregation that awards full credit exactly when all required assertions hold. No particular partial-credit values are required.

These obligations identify the trusted primitive implementations and adapters. They are not consequences of type checking alone.

Proof of Theorem 2. Unroll bounded iterations on the fixed finite instance and evaluate the resulting dependency graph in topological order. By total deterministic primitive semantics, each node has a unique result given its predecessors. Thus solve mode terminates with a particular d^* and W^*.

Next apply induction over the reference tool-call composition. The initial concrete workspace represents the demanded world. By faithful tool mappings, each read returns the specified value and each sink realizes its abstract effect, with intermediate bindings preserved. The reference therefore reaches the required projection $q_P(W^*)$. By Eq. 13, all exported assertions hold there; Eq. 14 then gives reference score 1.

For any $\tau \in \mathcal{Y}_{\pi(w)}$, the full-credit condition and exact component export give

$$\begin{aligned} g_V(\tau) = 1 \quad &\Longleftrightarrow \quad \bigwedge_{i} a_i(W_\tau) \\ &\Longleftrightarrow \quad \bigwedge_{k=1}^{K} \left[q_k(W_\tau) = q_k(W^*) \right] \\ &\Longleftrightarrow \quad \tau \in A^*(w). \end{aligned}$$

The middle equivalence is the complete component export condition. This proves both inclusions in $A_V = A^*(w)$ and hence zero false acceptance and false rejection relative to the specified contract. □

What the proof does not require. L2 and L7 are useful checks on task structure, but are not needed for the set equality once the observable contract is fixed and completely exported. An unused input does not itself create a second correct answer. Likewise, an adjudication with no observable consequence does not violate extensional correctness; it fails the intended difficulty design. Even a syntactically connected adjudication may be masked by rounding or a later branch. Behavioral differencing checks these effects on the sampled instance.

What the proof does require. Negative assertions cover only their declared domain. Enumerating unselected recipients from one routing table cannot prohibit deliveries to every address outside that

table. If global absence of extra deliveries is part of the task, the contract needs an exact assertion over the complete relevant sent-message set. Similarly, a final-state equality check cannot detect a write subsequently undone unless the environment retains its history. These are changes to the observable contract and adapter, not consequences of co-compilation by itself.

A.4 Expressiveness and decision boundaries

Proposition 1 (Representation relative to the primitive basis)**.** *Any finite acyclic workflow whose queries, selections, computations, branches, and observable effects are compositions of the supported Prax primitives has a representation in Prax's decision-program language, including workflows with explicitly bounded iteration.*

Proof. Expand bounded iteration and topologically order the workflow graph. Emit the primitive implementing each node and bind each input to its predecessor's output. The resulting program has the same dependencies, concrete semantics, and observable effects by construction. □

This proposition specifies coverage relative to a fixed basis. It does not assert that arbitrary finite decision rules or arbitrary natural-language tasks are expressible. Operations outside the basis need new primitive contracts and service adapters. Unbounded iteration is excluded; a fixed-point computation with a known finite bound can still be unrolled if its operations are supported.

Proposition 2 (Decidability of finite decision consistency)**.** *Fix an instantiated world and a finite program. Let $\Phi_{P,W_0}(d)$ be a decidable constraint encoding of its declared decision rules, with every decision variable ranging over an explicitly finite domain. Given a satisfying decision vector d^*, the existence of a distinct satisfying vector is decidable. So is the existence of one producing a different required observable projection.*

Proof. Enumerate the finite Cartesian product of decision domains, evaluate Φ_{P,W_0} on every assignment, and compare satisfying assignments with d^*. For the second question, evaluate their bounded downstream computations and compare the resulting projections. Both searches terminate. □

The world assignment is fixed in this proposition. Different assignments of names or amounts are different task instances, not competing solutions to one task. Moreover, uniqueness under the encoded rules is distinct from recovering those rules from prose. A counterfactual catalogue that changes one decision at a time is not an exhaustive search over all joint assignments. It can validate each listed mistake without constituting a global uniqueness certificate. Decidability also provides no polynomial-time or constant-cost bound.

Proposition 3 (Unbounded candidate domains can make uniqueness undecidable)**.** *If a constraint language admits candidates in $\mathbb{N}$ and predicates that simulate a machine for a candidate-specified finite number of steps, uniqueness of a satisfying candidate is undecidable, even when one satisfying candidate is supplied.*

Proof. Given a machine M and its input, define $\Phi_M(0)$ to be true. For $t \geq 1$, define $\Phi_M(t)$ to be true exactly when M halts within t steps. Every individual predicate evaluation terminates by bounded simulation, and 0 is a known satisfying candidate. It is the unique satisfying candidate exactly when M never halts. A general uniqueness decider would therefore decide the halting problem, which is impossible. □

This result shows why the finite-domain restriction matters for a general guarantee. It does not imply that all infinite-domain constraint fragments are undecidable. Nor does it turn synthesis into constant-cost computation: each generated instance still requires constraint solving, rendering checks, and replay. The practical benefit is reuse of the primitive contracts and checking infrastructure across instances, with residual auditing still needed for their implementation and linguistic realization.

A.5 Estimating ambiguity and evaluating verifier errors

Retaining the sampled specification supplies a correctness label for a candidate under that specification. It does *not* supply the posterior μ_x over all specifications that could have produced the same publication. Exact computation of $\mathrm{Amb}(x)$ therefore requires an explicitly enumerable generator or another justified inference procedure; it does not follow from knowing a single sampled w.

A controlled test of Theorem 1 can use finite families such as the rate-convention example, with known priors and intentionally identical publications. Enumerating their contracts and candidate outcomes makes the bound directly computable. This diagnostic construction should be distinguished from the training corpus, whose intended criteria must be recoverable from the environment.

For ordinary compiled instances, verifier errors can instead be measured on independently labeled candidate sets: the reference, valid alternative trajectories, declared mistakes, combined mistakes, and unsolicited side effects. Give a post-hoc verifier constructor only (m, W_0) and evaluate both its output and Prax's verifier on the same candidates. Reference replay alone does not test false rejection of alternative solutions; single-site mutations alone do not test arbitrary reward exploitation. Candidate labels should be checked against primitive-level contracts and independently replayed, rather than assigned solely by the verifier being evaluated. Report uncertainty and the composition of the candidate set. Changes in empirical error under presentation or decision-count interventions are experimental questions, not unconditional monotonicity predictions of the two theorems.

B Intent and language details

B.1 Intent enumeration and diversity controls

Graph construction. Artifacts carry one of three roles: SOURCE, which can only head a path; JUNCTION, which is a semantically specific intermediate artifact; and TERMINAL-ONLY, which includes generic deliverables that would otherwise connect unrelated skills. The junction predicate permits same-domain composition, shared technology entities, or a declared workflow hand-off such as development to testing. Same-family repetition is restricted to avoid chains built from near-duplicate skills.

Enumeration and quotas. Dynamic programming counts legal paths in each occupation–depth cell. Small cells can be exhausted; large cells are sampled backward from terminal artifacts through legal predecessors. Endpoint quotas and path fingerprints control repetition. Enumeration covers admissible graph structures, not all semantically possible workflows. Acceptance failures can alter the realized distribution, so the ledger tracks delivered instances and compilation rejection rates as well as requested quotas.

Intent forms and slots. The intent generator distinguishes single-capability requests, multi-hop chains, parallel goals, ambiguous routing resolved by environmental evidence, and negative requests outside the capability library. Only forms with a supported observable success condition enter the compiled corpus. For example, a negative request needs a specified no-action or recorded-response contract; recognizing that no library skill applies is not by itself a machine-checkable workspace task.

Eight slot groups control deliverable specification, process topology, process constraints, interaction mode, material condition, user persona, surface form, and goal clarity. Structural slots follow the capability path; other slots are sampled subject to compatibility rules, usually activating one to three non-default groups. Specificity distinguishes general requests, named capability features, and edge cases. These are separately configurable dimensions, not a statistical independence assumption. The compiled setting also excludes requested interactions or process constraints that the environment cannot represent.

Rendering and coverage audit. Prompt blocks combine the capability context, situation structure, narrative focus, specificity, and active slots. The resulting request should not quote skill names or enumerate the intermediate capability sequence. A retrieval check compares the generated request with its intended skill path, recording partial recall as a diagnostic for inspection; retrieval agreement is not a proof of semantic validity. The accepted program is checked against required service

bindings, primitive families, and the decision budget before its instance is credited to the coverage ledger.

B.2 PRIMITIVE INVENTORY AND STATIC CHECKS

Each primitive implements the five-part contract described in §2.3. The v0 inventory contains the following 23 primitives.

Sources (five). `find_target` identifies an entity among confusables; `read_policy` retrieves governing rules; `lookup_row` performs a keyed lookup; `traverse_ref` follows a declared relation; `extract_from_text` extracts a specified fact from a supported textual encoding. Their materialization requirements include the necessary records, closed references, and identifiable facts.

Adjudication (five). `resolve_latest` selects by temporal validity and recency; `resolve_authority` applies source precedence; `resolve_tie` applies an explicit tie-break; `match_fuzzy` disambiguates a noisy key; `check_window` tests temporal eligibility. Each declares finite alternatives and the evidence needed to distinguish them. Confusables, stale values, tied candidates, and boundary cases supply targeted incorrect choices.

Computation (five). `compute_arith` evaluates an arithmetic operation; `aggregate` combines a filtered set; `score_weighted` applies a policy-specified score; `threshold_filter` selects records by a predicate; `format_value` applies a declared presentation rule. The sampler respects definedness, numeric precision, rounding, and downstream distinguishability where relevant.

Control and sinks (eight). `branch_policy` selects a policy route; `check_exception` evaluates an exception clause; `foreach` iterates over a bounded collection; `guard_empty` handles an empty result explicitly. `write_record`, `notify`, and `log_note` declare record, delivery, and note effects; `no_action` declares an absence-of-effect contract over a specified scope, optionally with a recorded explanation.

The static checks are:

- L1. Check types and distinguish allowed schema/action constants from sampled instance values.
- L2. Require every task-relevant source and intermediate value to reach a sink. Independent background distractors are introduced by the renderer, not as unused program computations.
- L3. Require an assertion mapping for every sink on its bound service.
- L4. Check required primitive families and the declared decision budget.
- L5. Check the specified diversity of decision mechanisms across repeated targets.
- L6. Require explicit distractor and scope declarations for empty-set and no-action tasks.
- L7. Require a dependency path from every adjudication result to a graded sink argument.
- L8. Restrict calls to supported tools and check read/write bindings and reference closure.

Bounded iteration, finite candidate domains, and explicit tie resolution are language-level restrictions. L2/L7 provide structural diagnostics; whether a declared mistake changes a score is checked after instantiation. Program repair uses a bounded number of diagnostic rounds, with unsuccessful attempts counted toward synthesis cost.

B.3 ACCEPTANCE RECORDS AND DIFFICULTY PARAMETERS

Every accepted instance retains its sampled assignment, configuration, accepted rendering artifacts, assertion weights, reference replay, and counterfactual results. Program fingerprints normalize incidental names while retaining primitive composition, dependencies, and service bindings. They can reject duplicate skeletons or cap their multiplicity; re-instantiating an admitted skeleton under an explicit quota remains a supported scaling mechanism. Structural deduplication and intentional within-family variation are separate controls.

For Prax's $n \geq 1$ Boolean assertions a_i with weights $\lambda_i > 0$, the concrete scoring rule is

$$g_V(\tau) = \frac{\sum_{i=1}^{n} \lambda_i a_i(W_\tau)}{\sum_{i=1}^{n} \lambda_i}. \tag{15}$$

This instantiates Eq. 1 and satisfies the full-credit condition in Eq. 14. For J counterfactual trajectories, store the failure matrix and corresponding scores

$$M_{ji} = 1 - a_i(W_{\tau_j^-}), \qquad s_j = g_V(\tau_j^-) = 1 - \frac{\sum_i \lambda_i M_{ji}}{\sum_i \lambda_i}. \tag{16}$$

Every row must contain a failure. Different rows identify different violated obligations even when the scalar scores coincide. An optional stricter filter additionally requires pairwise distinct s_j to separate mistakes by scalar reward. This criterion is independent of the exactness claim in Theorem 2. Correlated mistakes can legitimately share a failure vector or score. Experiments should report which filter is enabled and its rejection rate rather than infer error coverage from scalar separation alone.

The structural difficulty prior counts adjudication sites and reference traversals, with additional weight for operators such as aggregation, empty-set guards, and tie resolution. These initial weights are calibration heuristics. Presentation controls include structured-field exposure (E0), direct textual statements (E1), evidence split across carriers (E2), and supported encodings requiring extraction or conversion (E3). Trigger specificity ranges from explicit instructions (L0) through policy references (L1) to event-only requests (L2), the last requiring accessible policy evidence sufficient to determine the required action. World size and distractor count control search burden under the tool budget.

A configuration change is valid only when the program and adapter support the resulting encoding. In particular, E3 may require extraction or conversion operations and cannot be assumed to preserve an arbitrary program unchanged. Calibration uses observed success rates and assertion failures to adjust the structural prior. Failure concentration is a screening heuristic; ambiguous requests and systematic verifier defects can themselves produce concentrated errors and require direct inspection.

C END-TO-END CASE STUDY

Consider Figure 4. The request asks the agent to close a renewal opportunity and notify the team according to the current policy. The workspace includes three near-name decoys, a GBP/USD table, a customer-tier table, a stale CRM tier field, and a policy email. To make the task answerable, the published records identify the target and specify which timestamp governs rate selection, that the tier table is authoritative, and which notification route applies to each tier.

Listing 1: Full decision program for the compilation example in Figure 4.

```
def program(w, trig):
    opp = find_target(w.sf.opportunities,
                      key_hint=trig.subject_hint, n_confusables=3)
    policy = read_policy(w.gmail,
                         rules=["routing_table", "amount_format"])
    rate = resolve_latest(w.sheets["fx"], key=opp.currency)
    tier = resolve_authority(
        fresh=lookup_row(w.sheets["tier"], key=opp.account),
        stale_carrier=opp.account.tier)
    amount = format_value(compute_arith(opp.amount, "*", rate),
                          spec=policy.spec.amount_format)
    write_record(opp, fields={"stage_name": "Closed Won"})
    notify(to=branch_policy(policy.table.routing, on=tier),
           body=[opp.name, amount])
```

Demand ledger and instantiation. The target is "Aurora Systems – Renewal", with amount £84,000, alongside decoys such as "Aurora Solutions – Renewal". The two applicable FX rows contain 1.25 dated March 2 and 1.31 dated August 14. The task occurs after both dates, and the policy selects the latest applicable row. The authoritative tier is Tier-1; the CRM residue is Tier-2. The policy routes Tier-1 to `vp-emea@corp.example` and Tier-2 to `regional-desk@corp.example`, with a third unselected route in the declared candidate domain. Policy-specified formatting requires the currency symbol, grouping separator, and two decimal places.

Solve mode and assertions. The computed amount is $84{,}000 \times 1.31 = 110{,}040$, formatted as "$110,040.00". The selected route is `vp-emea@corp.example`. Representative exported predicates are:

```
field_equals(OPP-3412, stage_name, "Closed Won")
sent_to_with_body_contains(vp-emea@corp.example,
  ["Aurora Systems", "$110,040.00"])
not_sent_to(regional-desk@corp.example)
```

The complete instance includes a negative delivery assertion for each other unselected route. These predicates admit different message wording around the required content. They do not require byte equality with the reference email, nor do they exclude every possible unrelated side effect unless that prohibition is added to the contract.

Reference replay. The reference searches the opportunities, reads the applicable policy and tables, resolves the target, rate, and tier, then updates the opportunity and sends the notification. Its instantiated tool sequence must execute successfully and satisfy all assertions. Equivalent trajectories may retrieve the policy and financial data in a different order.

Counterfactual replay. Selecting the old rate produces $105,000.00 and violates the content assertion. Trusting the CRM tier routes the message to the regional desk, violating both the required delivery and the prohibition on that route. Selecting a near-name decoy fails the target-update assertion and may also fail the content check. A wrong policy route is tested separately from the stale-tier cause, even if it induces the same visible error. Each counterfactual must lose score; distinct causes are not assumed to produce distinct scalar scores. Combined errors and deliveries outside the declared routing domain are useful additional probes of the verifier's scope.

D TRAINING DATA COVERAGE

We measure tool-use coverage in 65k training records by mapping recorded native-tool invocations and recognized API endpoints to tool families and functional capability domains. Each record contributes at most once to each family or domain, regardless of the number of calls. API discovery and agent-bookkeeping calls are excluded. Of the remaining invocations, 99.32% are mapped; the 541 records without a mapped invocation remain in the denominator. This analysis describes observed tool use, including attempted calls and trajectory segments; it does not count unique tasks or program skeletons.

E CONTAMINATION ANALYSIS

We assess potential content contamination using 8-gram overlap in both task prompts and tool-accessible workspace text. The recorded audit covers 65k retained synthetic tasks, each compared with its designated source or donor from Benchmarks' test set. Production checks use lowercased word tokens and reject instances whose 8-gram overlap exceeds 5% in either channel. Additional checks detect reuse of selected source assertion values and reject filtered workspace content-string overlap above 10%. Replaying these checks yielded **zero violations** among the audited tasks (Table 3), providing evidence of limited content reuse under the specified checks.

Table 3: **Benchmark content-overlap audit**

Check or statistic	**Recorded result**
Retained tasks violating production filters on replay	0 / 65k
Source–task request overlap: median / 90th percentile / max	0.0% / 1.4% / 8.6%
Source–task request overlap above 5%	0.2% of tasks
Cross-skeleton pairs with overlap above 15% (400-task sample)	0

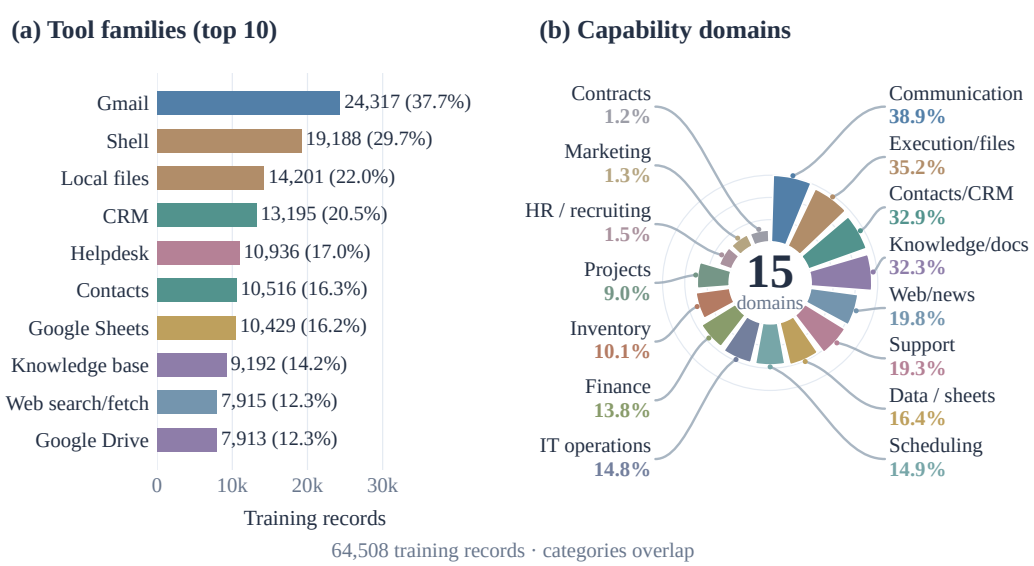


Figure 6: **Tool-use coverage of the training data.** Left: the ten most prevalent invoked tool families, with record counts and percentages. Right: coverage of 15 functional capability domains derived from an explicit tool mapping. Percentages use all 64,508 training records; categories overlap. Radial bar length is proportional to the square root of record prevalence, and shared colors indicate the same domain across panels.

Table 4: Estimated cost per task for each model.

Model	Cost per Task (USD)	Avg. Score
Qwen3.6-35B-A3B	0.0518	32.3
Occamy-1.0 35B-A3B	0.0712	47.0
Ornith-1.5-35B-A3B	0.0802	42.3
Nex-N2.5-mini 35B-A3B	0.0809	33.5
Apodex-1.1-mini 35B-A3B	0.0862	36.8
Minimax-M3	0.1677	44.8
Ornith-1.5-397B	0.2691	48.1
Kimi-K3	0.5452	56.5
GLM-5.3	0.5989	61.6
GLM-5.2	0.6241	47.6
Qwen3.8-Max	0.6773	57.4
Prax-35B-A3B	**0.1418**	**56.0**

F Detailed Cost

We report the estimated inference cost per task based on model input and output token usage, as shown in Table 4.