Ask Now, Use Later: Benchmarking the Proactivity Gap in Long-Lived LLM Agents

Anonymous ACL submission

Abstract

A long-lived LLM agent, such as OpenClaw, earns its value by acting on a user's preferences and constraints across sessions, not just the current request. Yet today's agents keep what a user volunteers but rarely ask for what stays unspoken, leaving a **proactivity gap in long-lived LLM agents**: an agent cannot act on a preference it never obtained. As users delegate more of their affairs to agents, the impact of this gap grows. We isolate one concrete, controllable slice of this gap as **Ask-to-Remember (ATR)**: the agent decides whether to ask now for a reusable user preference that the current task does not need but a later session with the same user will. ATR is hard even to evaluate: the right question is underdetermined and its payoff deferred to tasks that may never arise. **ATRBench**, to the best of our knowledge the first ATR benchmark, makes it measurable by fixing each user's preferences as hidden ground truth, so success demands asking, not recall. Across eight frontier LLM agents, defaults fall at least 62 points below an oracle handed the relevant preference, and prompting closes little of it. Diagnostics identify acquisition as the bottleneck. ATRBench surfaces this proactivity gap in current agents and offers a diagnostic testbed for closing it.

1 Introduction

LLM agents are increasingly developed as long-lived personal assistants that serve the same user across many sessions, from research systems (Park et al., 2023; Packer et al., 2023; Zhong et al., 2024; Brady, 2026) to open-source runtimes such as OpenClaw.[1] The value of such an assistant lies not in handling any single request in isolation, but in accumulating user information over time and acting on it in later tasks. Such information includes what the user prefers, what standing constraints they hold, and how they want tasks carried out. As

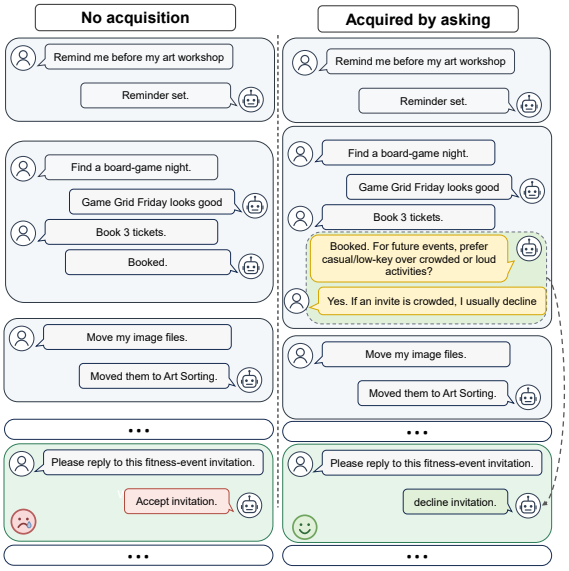

Figure 1: Passive memory vs. Ask-to-Remember over four sessions. The user never volunteers the rule *avoiding crowded or loud events*: passive memory misses it and later accepts a crowded invitation, whereas ATR elicits the rule during an earlier related task and later declines correctly.

these agents take on recurring matters on the user's behalf, such as scheduling, travel, communication, and shopping, personalization turns from a convenience into a precondition for reliable action. An assistant that does not know the user's standing preferences may carry out a request to the letter and still act in a way the user would not accept.

Figure 1 illustrates the failure mode this paper studies: the agent handles the user's early requests, but never asks about a latent preference and later acts against it when the user is absent. This is not a failure of task execution but of acquisition, since the agent never obtained the reusable preference the later decision required. Passive personalization preserves only information the user has already revealed, leaving unspoken but future-relevant preferences outside the agent's usable user model. We

[1] https://openclaw.ai/

call this the **proactivity gap in long-lived LLM agents**: the agent does not proactively acquire latent user information before it is needed. Existing evaluations typically either assume the relevant user information is already available, as in profile-, history-, or memory-based personalization benchmarks (Salemi et al., 2024; Maharana et al., 2024; Zhao et al., 2025; Lyu et al., 2026; Duan et al., 2026), or study questions that serve the current task, as in clarification and planning settings (Xu et al., 2019; Zhang et al., 2024); neither setting exposes acquisition for later use.

As these assistants increasingly act on the user's behalf, including while the user is offline, the proactivity gap grows more consequential. Acquisition can take many forms; we focus on explicit asking, a channel the agent directly controls that carries a real interaction cost and leaves a traceable answer, which makes it both concrete and observable. We cast asking for future-relevant user information as a new task paradigm, **Ask-to-Remember (ATR)**: *the agent decides whether to ask a question whose answer is not needed for the current task but can be reused in a later session with the same user.* ATR is hard for two reasons: (i) the asking target is underdetermined, since the useful questions probe the user's latent standing preferences, which are unknown to the agent and not singled out by the current task; and (ii) the payoff is delayed and uncertain, since a question's value surfaces only in later tasks that may or may not arise. Together these leave the agent little to anchor on when deciding whether and what to ask. The same two properties also make ATR hard to evaluate directly: there is no ground truth for which question the agent should have asked, nor an observable payoff for one it skipped. Long-term utility over a realized interaction history then entangles asking with how the agent later stores and uses information, so asking cannot be cleanly isolated.

To make ATR empirically testable, we build **ATRBench**, a controlled benchmark that turns each difficulty into a measurable form while leaving the agent a genuine ATR task to solve. For the underdetermined target, ATRBench fixes the user's latent standing preferences as episode-level ground truth, visible to the evaluator but hidden from the agent, and embeds the agent in ordinary interaction contexts under the same persona that never disclose those preferences, so it preserves natural opportunities to ask while still demanding acquisition rather than recall. For the delayed payoff, it binds each preference to a future task whose correct behavior depends on it, turning a question's value into a directly checkable later outcome. Evaluation can then separate acquisition through earlier asking from correct application of the acquired information. Across eight frontier LLM agents, oracle access to the relevant preference raises future-task accuracy by 62.1–76.6 points, but prompting recovers at most 15.5% of this gap.

Our contributions are as follows:

- To the best of our knowledge, we are the first to define the **proactivity gap in long-lived LLM agents**. The gap grows more consequential as agents act more autonomously on the user's behalf, including while the user is offline and unable to correct them.
- We formalize **Ask-to-Remember (ATR)**, an ask-now/use-later task that instantiates explicit asking as one concrete slice of the proactivity gap. We introduce **ATRBench**[2], a controlled benchmark for evaluating LLM agents on ATR by withholding target preferences during learning and scoring later preference-bound tasks.
- We systematically evaluate eight frontier LLM agents and find a large proactivity gap: their proactive acquisition ability is poor, and prompting-based interventions yield only small gains. ATRBench offers the community a testbed to guide research on closing this gap.

2 Problem Formulation

ATR formalizes the explicit-asking slice of the proactivity gap (§1), the failure to acquire latent reusable user information before it is needed. We formalize the task (§2.1) and then derive what a controlled evaluation of it requires (§2.2).

2.1 Task Formulation

Consider an agent serving the same user u across sessions $t = 1, 2, \ldots$. The user holds a latent state I^* of stable preferences, constraints, and default choices, and later tasks are drawn from a user-specific distribution $\mathcal{D}_u$. Long-term personalization is measured by how well the agent's actions across the session sequence conform to I^*, not by literal completion of any single task τ_t.

[2] https://anonymous.4open.science/r/ATR-anon-5803/README.md

The agent's usable user model $\hat{I}_t$ is its current estimate of I^*, and the user-model gap is the part of I^* still missing from it. A cross-session substrate T carries each interaction history h_t into later state, so passive personalization narrows the gap only as the user volunteers information. While the user is online, ATR instead lets the policy π ask a question q_t whose answer is not needed for τ_t but targets a reusable part of I^* relevant to future tasks in $\mathcal{D}_u$. The answer then enters h_t and propagates through T into $\hat{I}_{t+1}$, so the agent closes the gap itself before the information is needed, rather than waiting for the user to reveal it.

Asking is not free: it carries an interaction cost $C(\pi)$, while its benefit appears only through future-task utility $U(\tau; I^*, \pi)$ evaluated against the true I^*, giving the objective

$$V_{\text{util}}(\pi) = \mathbb{E}_{\tau \sim \mathcal{D}_u}[\, U(\tau; I^*, \pi)\,] - C(\pi),$$

a utility–cost view shared with preference elicitation (Boutilier, 2002; Martin et al., 2024; Choudhury et al., 2025). Two challenges follow: the current task does not reveal which parts of $I^* - \hat{I}_t$ are worth asking about, so the target is underdetermined; and the agent does not know which tasks $\mathcal{D}_u$ will draw, so the payoff is delayed and uncertain.

2.2 Why ATR Resists Direct Evaluation

The two challenges also block any direct evaluation: an evaluator observes neither I^* nor which tasks $\mathcal{D}_u$ will draw, and a realized trace folds the asking policy π together with the substrate T, so no trace isolates the acquisition ATR sets out to measure.

A valid evaluation must therefore supply controlled counterparts of these quantities, known to the evaluator but withheld from the agent. The evaluator fixes the reusable information as standing preferences known to it but hidden from the agent, so success requires acquisition rather than recall; binds each preference to a concrete future task whose outcome depends on it, making the delayed utility $U(\tau; I^*, \pi)$ directly checkable; and holds the substrate T fixed across agents, so differences reflect asking rather than memory machinery. Decomposing performance into asking, acquisition, and application then attributes a low score to proactive acquisition itself rather than to an unsolvable task, a weak substrate, or mere recall. ATRBench instantiates this design in §3.

3 ATRBench

ATRBench instantiates the three requirements of §2.2: it realizes I^* as evaluator-known but agent-hidden standing rules, the delayed payoff as rule-bound future tasks, and a fixed cross-session substrate so that differences reflect asking rather than memory machinery. Each episode accordingly pairs one persona's hidden rules with learning sessions, in which the agent may ask, and test sessions, in which the resulting behavior is scored (Figure 2). The rest of this section defines the protocol (§3.1), its metrics (§3.2), and its construction (§3.3).

3.1 Episodes and Protocol

An ATRBench episode centers on one long-term user u and three coupled parts: a hidden standing-rule pool $\mathcal{R}_u$, an ordered sequence of learning sessions $\mathcal{L}_u$, and a set of test sessions $\mathcal{T}_u$. A binding map $\beta_u : \mathcal{T}_u \to \mathcal{R}_u$ assigns each test session to the hidden rule that determines its correct behavior. A standing rule $r \in \mathcal{R}_u$ represents a stable preference or constraint that the user does not volunteer. Each rule has a canonical answer a_r (the first-person reply when asked) and a required later behavior ρ_r (what the bound test session must satisfy), connecting a natural-language preference to a checkable later decision while leaving it hidden unless the agent asks.

The learning phase runs first. In each learning session, the user is online, played by a fixed LLM simulator that holds the persona and the current task but is blind to the scaffold's matching decision, so it cannot leak which messages were treated as rule asks. Hidden rules are not in its prompt and are not volunteered by default. The agent completes the current task through normal interaction and may also ask candidate ATR questions; after each learning session, the cleaned transcript is appended to a cross-session context block that becomes part of the next session's state. Deciding whether and what to ask in these sessions is exactly the underdetermined, delayed-payoff decision of §2.1.

When the agent sends a user-facing message, it exposes a visible output o delivered to the user, together with an internal reason field z that the Router consumes as auxiliary intent; z is withheld from the simulator, Classifier, evaluator, and frozen context. A fixed Router–Classifier scaffold (Figure 2) then processes the message in two LLM stages. The Router's output is $(b, q) = \text{Router}(o, z)$: a

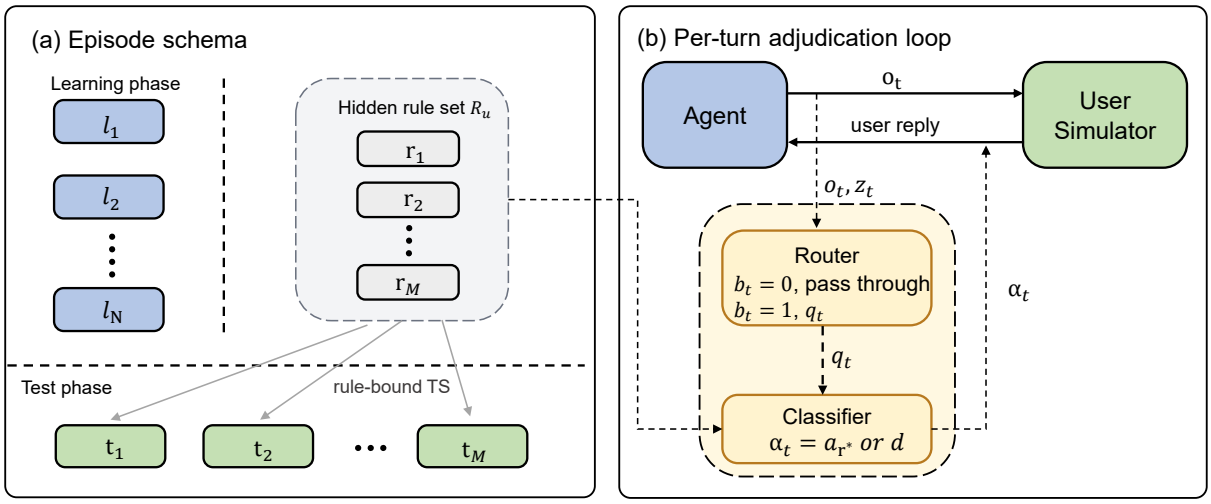

Figure 2: Two-phase ATRBench protocol separating user-online learning from user-offline testing. (a) Each episode runs the learning phase, freezes the accumulated cross-session context, and evaluates test sessions from that frozen context. (b) On each learning turn, the agent and user simulator form the main dialogue loop; the Router–Classifier scaffold acts as an optional side-channel adjudicator, passing Router-negative turns through unchanged and, on Router-positive turns, substituting the simulator's marked reply with a matched first-person rule answer or a no-rule deflection.

binary verdict $b = 1$ iff o asks a strict standing-rule question, and an extracted span q. When $b = 1$, the Classifier $\mathrm{cls}(q)$ returns either a hit $r^* \in \mathcal{R}_u$ or $\varnothing$ when no rule matches. These verdicts set the injection token delivered to the user simulator,

$$\alpha = \begin{cases} \varnothing & b = 0, \\ a_{r^*} & b = 1,\ \mathrm{cls}(q) = r^*, \\ d & b = 1,\ \mathrm{cls}(q) = \varnothing, \end{cases}$$

where d is a fixed deflect phrase and $\alpha = \varnothing$ means no injection, so the simulator answers normally. Both modules are frozen during evaluation and manually validated on held-out samples outside the scored episodes (Appendix C). The scaffold also records whether the agent asked, which rule it acquired, and which transcript evidence can later enter the frozen context.

After the learning phase, ATRBench freezes the accumulated cross-session context and evaluates test sessions independently. In the test phase, the user is offline: the agent cannot ask further questions and must act using only the current test instruction and the frozen context. Each test session $t \in \mathcal{T}_u$ is tied by $\beta_u(t)$ to one hidden rule and succeeds only if the agent's behavior satisfies $\rho_{\beta_u(t)}$. Appendix B gives the agent interface, simulator mechanism, and runtime orchestration details.

3.2 Scoring and Diagnostic Variants

The primary metric is **TSAcc**, the success rate on user-offline test sessions:

$$\mathrm{TSAcc}(u) = \frac{\left|\{\, t \in \mathcal{T}_u : \mathrm{pass}(t, \rho_{\beta_u(t)}) \,\}\right|}{|\mathcal{T}_u|}.$$

The predicate $\mathrm{pass}(t, \rho_{\beta_u(t)})$ checks whether the agent's first relevant decision in test session t satisfies the required behavior for the bound rule. This check is a deterministic match against the rule's tool-level binding, computed programmatically from the trajectory rather than scored by an LLM judge (Appendix D.2). We use first relevant decision rather than eventual recovery because the offline user cannot correct the agent after an initial violation. All metrics are computed per persona and then macro-averaged across personas, treating the persona as the replication unit; Appendix D.1 gives the full aggregation recipe.

ATRBench also reports learning-side diagnostics. **RuleAsk** is the number of strict standing-rule questions per learning session. **RuleCov** is the fraction of hidden rules hit at least once during learning, measuring acquisition breadth. **AcqPrec** is the share of strict rule questions that the Classifier maps to a hidden rule, measuring whether questions target useful future information. **AppliedRate** is the test-session pass rate over learnable rules that were Classifier-confirmed as acquired during learning, separating acquisition failure from later-use

Variant	Guidance	When	What
`default`	none	free	free
`atr`	rule	free	free
`always_ask`	rule	fixed	free
`oracle`	answer	—	—

Table 1: Diagnostic variants. *Guidance*: *none* = no rule-asking content; *rule* = generic definition of a rule plus a cost–benefit reminder (discloses no specific rule); *answer* = direct injection of the TS-bound rule's canonical answer. *When*/*What*: whether the asking moment / target is fixed or free; `oracle` skips LS, so neither applies.

failure. Together, these metrics decompose failure into whether the agent asks, whether its questions acquire future-relevant information, and whether acquired information is later used.

We evaluate four diagnostic variants while keeping the episode data fixed. The `default` variant measures spontaneous ATR behavior without ATR-specific instruction. The `atr` variant gives generic guidance that the agent may ask about reusable standing rules, but does not reveal any specific rule. The `always_ask` variant further fixes the asking moment, helping distinguish not asking from asking imprecisely. The `oracle` variant, corresponding to `oracle_target` in Appendix B.5, bypasses learning and supplies the test-bound rule answer, giving an application ceiling for TSAcc. The gap from `default` to `oracle` estimates how much executable payoff is available if the relevant preference is known; non-oracle gap recovery estimates how much of that payoff the agent recovers through its own asking behavior. Appendix B.5 details the variant prompt blocks.

3.3 Construction and Quality Control

ATRBench is built around a decoupled construction principle: standing rules, rule-bound test sessions, and learning sessions are generated from the same persona and environment, but target rules are not directly inserted into the learning trajectory. This prevents the benchmark from collapsing into passive transcript use. At the same time, learning sessions remain grounded in the same user's everyday tasks, so the agent has natural opportunities for explicit asking about reusable preferences. All construction stages use frontier models available at the time of the study: GPT-5.4 throughout, with Gemini 3 Flash Preview for test-session quality control. The main text focuses on construction validity. Construction mechanics, runtime and error handling, scaffold calibration, and prompt interfaces are deferred to Appendices A–I.

Standing rules. For each persona sampled from Nemotron-Personas-USA, we generate candidate standing rules conditioned on the persona profile and the benchmark's personal-assistant environment. Each candidate must express a stable preference or constraint, map to a checkable later behavior, and be counter-default: a rule-blind helpful agent should plausibly take a different action, the *counterfactual default*. Candidates are screened by a counter-default check and a binding-soundness check. Across the 20-persona ATRBench dataset, 496 candidates are produced and 340 pass both checks (a mean of 17.0 per persona); these form the candidate retained rule pool from which test sessions are then generated.

Rule-bound test sessions. For each retained rule, we generate a future test session by working backward from the required rule-bound behavior. The session is designed so that a rule-aware agent can satisfy the required behavior, while a rule-blind agent is pulled toward the counterfactual default. Each candidate is screened by a paired upper-bound and lower-bound check, each evaluated over three independent simulated-agent samples. The upper bound runs a rule-aware agent (the rule's canonical answer injected) and the lower bound a rule-blind agent (no rule); a bound is decided by a two-of-three majority reaching the gold action. A candidate passes only when the upper bound passes and the lower bound fails, so the session is solvable with the rule but discriminates against acting without it. Failed candidates are refined with the QC trace for up to two rounds before being dropped. After this stage, the scored rule pool contains 284 rule–test-session pairs, giving a one-to-one pairing between final rules and user-offline test sessions.

Rule-blind learning sessions. For each persona, we generate everyday learning sessions from the persona and task domains, but without directly conditioning them on the target standing rules. A skeleton stage samples temporally ordered session themes; a fill stage instantiates each theme into a task with references and an oracle trajectory. Each filled session is validated by a dry run in the executable environment. We then select $2|\mathcal{T}_u|$ learning sessions per persona, prioritizing sessions whose normal task context shares action surfaces with the corresponding test pool while preserving generated temporal order. By construction, $205/284 = 72.2\%$ of the standing rules have at least one selected learning session whose oracle

Scope & Scale		*Domain (rule count)*	
Personas	20	Travel	74
Domains	6	Reservation	56
Tools	74	Commerce	53
Rules	284	Scheduling	40
TS/persona	14.2 (11–19)	Workspace	33
LS/persona	28.4 (22–38)	Communication	28
Action surface (rule count)			
Object-ID parameter	124	Tool identity	59
Enum parameter	99	Confirmation	2

Table 2: ATRBench dataset statistics for the 20-persona cohort. The `confirm` surface yields only two retained rules; we report it for completeness but base no per-surface claim on it.

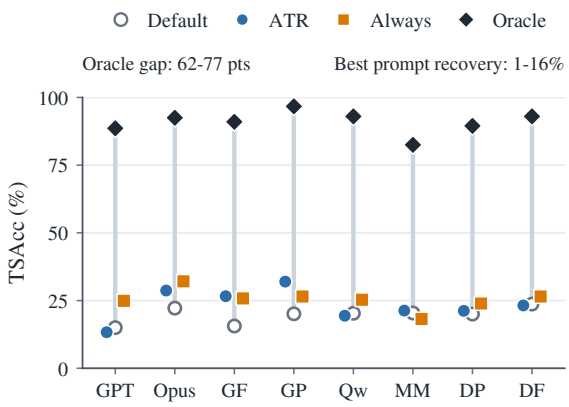

Figure 3: ATR and `always_ask` leave most of the executable oracle gap unclosed. Gray intervals connect `default` to `oracle`; blue circles and orange squares mark ATR and `always_ask` (abbreviations in §4.1).

trajectory touches the rule's action surface (per-persona range 55–91%), even though no session is conditioned on the rules. This indirect coverage gives the agent realistic service context for encountering rule-relevant decisions without making the future-relevant rules explicitly available.

Final dataset. The resulting cohort contains 20 personas, 284 standing rules, 568 learning sessions, 6 domains, and 74 tools (Table 2). Section 4 evaluates frontier LLM agents on this fixed protocol to measure how much delayed personalization payoff they recover through asking-based acquisition.

4 Experiments

4.1 Setup

Models. We evaluate eight frontier API LLMs as the agent under test: GPT-5.4 (GPT) (OpenAI, 2026), Claude Opus 4.7 (Opus) (Anthropic, 2026), Gemini 3 Flash Preview (GF) (Google DeepMind, 2026a), Gemini 3.1 Pro Preview (GP) (Google DeepMind, 2026b), Qwen3.6-Plus (Qw) (Qwen Cloud, 2026), MiniMax M2.7 (MM) (MiniMax, 2026), DeepSeek V4 Pro (DP) (DeepSeek-AI, 2026), and DeepSeek V4 Flash (DF) (DeepSeek-AI, 2026). We follow each provider's default decoding parameters and enable its reasoning or thinking-budget mode where exposed. Backend identifiers and provider pages are in Appendix Table 11.

Sweep and variants. We sweep the four variants in Table 1 across the 20-persona cohort of Table 2, producing $8 \times 4 \times 20 = 640$ (model, variant, persona) cells. Each cell runs one learning sequence and then all rule-bound test sessions for that persona, single-trial with a 20-turn cap per session. We report persona-macro results unless stated otherwise; detailed execution and per-model inference settings are in Appendix E.

Scaffold and shared infrastructure. Only the agent model varies across cells: user simulator and Classifier use GPT-5.4, Router uses Gemini 3 Flash Preview, and all cells share the same raw-context layer, retry policy, and evaluator.

4.2 Main Results

Finding 1: the evaluated frontier API agents leave a large ask-to-remember gap. In the primary scorecard (Figure 3, Table 3), rule-naive `default` agents solve only 15.0–23.7 % of rule-bound future tasks, whereas `oracle` agents reach 82.5–96.7 % when the rule answer is injected. This 62.1–76.6 point Gap shows the test sessions are usually executable once the missing rule is known; the hard part is acquiring it beforehand.

Finding 2: the non-oracle interventions barely engage the asking channel and close little of the gap. The best non-oracle TSAcc beats `default` by only +0.9 to +11.9 points, recovering 1.3–15.5 % of the oracle gap. The `atr` setting rarely activates rule asking: six of eight models post RuleAsk ≤ 0.14 per LS, and only the Gemini models ask at non-trivial volume (RuleAsk 0.51 and 0.80). Which of `atr` or `always_ask` wins is model-dependent (Table 3); we read this as a broad proactivity gap, not a ranking between the two settings.

4.3 Acquisition and Application Diagnostics

Figure 4 decomposes the non-oracle variants into asking volume, useful acquisition, and downstream application (per-model numbers in Appendix G.2). The failure is not merely whether models ask,

Model	TSAcc (%) ↑				Ask ↓		Cov (%) ↑		AcqPrec (%) ↑		App. (%) ↑		Gap
	Def.	ATR	Alw.	Orc.	ATR	Alw.	ATR	Alw.	ATR	Alw.	ATR	Alw.	
GPT-5.4	15.0	13.3	24.9	88.6	0.01	1.00	0.4	12.7	5.0†	9.4	0.0†	80.9	73.7
Opus 4.7	22.2	28.7	**32.1**	92.5	0.14	1.01	7.1	**15.6**	**22.4**	8.9	97.7	83.2	70.3
Gemini 3 Flash Preview	15.6	26.6	25.8	91.0	0.51	1.01	12.0	14.4	13.1	8.7	78.4	70.0	75.4
Gemini 3.1 Pro Preview	20.1	**32.0**	26.5	**96.7**	0.80	1.00	**13.3**	11.3	10.1	6.0	88.9	**87.2**	76.6
Qwen3.6-Plus	20.3	19.4	25.3	93.0	0.01	0.96	0.7	10.3	10.0†	8.1	100.0†	73.5	72.7
MiniMax M2.7	20.4	21.3	18.2	82.5	0.00	0.09	0.0	1.8	–	4.2	–	25.0†	62.1
DeepSeek V4 Pro	20.0	21.2	23.9	89.5	0.01	0.98	0.6	14.5	5.0†	8.6	50.0†	70.6	69.5
DeepSeek V4 Flash	**23.7**	23.2	26.5	93.0	0.05	0.97	2.1	13.3	16.7	**10.4**	**100.0**	71.4	69.3

Table 3: Main ATRBench scorecard (20-persona macro). Def./Alw./Orc. abbreviate `default`/`always_ask`/`oracle`; ATR denotes `atr`. TSAcc, Cov (RuleCov), AcqPrec, and App. (AppliedRate) are percentages; Ask is RuleAsk per learning session; Gap is Orc.–Def. in points. Blue/orange columns are `atr`/`always_ask` (Figures 3–4); gray columns mark baselines, bounds, and gap. Gold cells mark the best model per column. † marks rates from negligible acquisition; – marks no classifier-confirmed hits; daggered cells are excluded from best-cell marking.

Variant	Ask↓	Cov↑	App↑	GapRec↑	Active
ATR	0.19	4.5	73.6	4.8	2/8
Always	0.88	11.7	70.2	7.8	7/8

Table 4: Acquisition summary over completed models (20-persona macro). Ask = RuleAsk, measured per learning session; Cov = RuleCov; App = AppliedRate over learnable, classifier-confirmed acquired rules; GapRec is the share of each model's default-to-oracle gap (Table 3) recovered by the variant, $(\text{var} - \text{default})/(\text{oracle} - \text{default})$ in TSAcc. Cov, App, and GapRec are reported in %. Active counts models with Ask ≥ 0.5, the ideal one-question-per-future-rule budget implied by ATRBench construction.

but whether they ask for future-relevant rules and whether application survives once they are acquired (Table 4, cohort level).

Finding 3: forcing one ask per LS raises ask volume but not what-to-ask precision. Only two of eight models clear the Active threshold (RuleAsk ≥ 0.5) under `atr`, but `always_ask` pulls RuleAsk to roughly 1.0 for seven of eight (all but MiniMax M2.7, at 0.09). The resulting asks rarely hit standing rules: among the seven active `always_ask` models, AcqPrec sits at 6.0–10.4 % (MiniMax M2.7 remains at 4.2% with low ask volume), below the 10.1–22.4 % the four `atr`-engaged models reach spontaneously. RuleCov stays below 16 % of each pool, and GapRec under `always_ask` reaches only −3.6 to 14.1 %. What to ask remains the binding upstream constraint even after asking time is fixed.

Finding 4: in this sweep, forced asking coincides with lower application accuracy. Across the four models with non-trivial `atr` acquisition (Claude Opus 4.7, Gemini 3 Flash Preview, Gemini 3.1 Pro Preview, DeepSeek V4 Flash), AppliedRate under `atr` sits at 78.4–100.0 %: once acquired, a rule is typically applied on the first attempt. Switching the same four models to `always_ask` drops Ap-

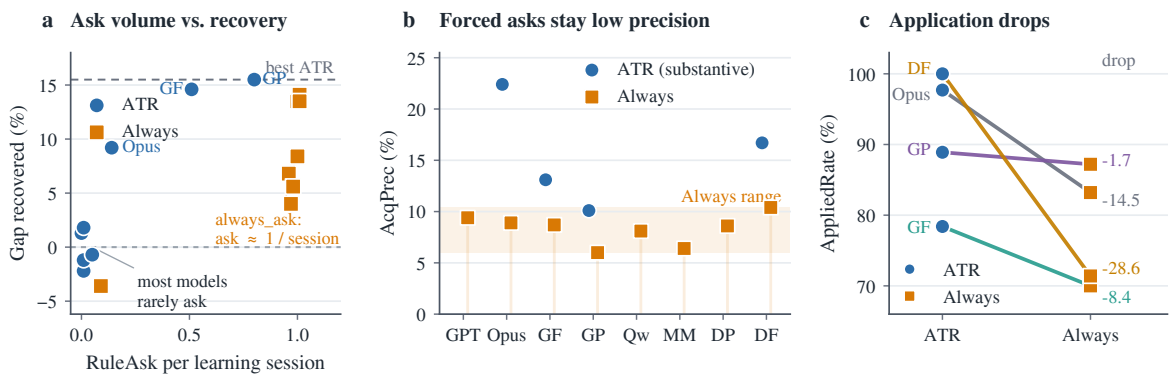

Figure 4: Forced asking raises ask volume but does not solve acquisition. (a) Higher RuleAsk does not proportionally recover the `default`-to-`oracle` gap. (b) `always_ask` precision stays low across models; sparse † ATR cells (Table 3) are omitted. (c) For models with non-negligible `atr` acquisition, `always_ask` coincides with lower AppliedRate. Abbreviations in §4.1.

Benchmark	Capability Dimensions				Eval. mode	Task target
	Long hist.	Pers. action	Proactive mode	Later-task eval		
Personalization benchmarks						
PrefEval (Zhao et al., 2025)	✓	×	passive	×	static	Long-context preference following
MPT (Yoon et al., 2026)	✓	✓	passive	×	static	Cross-session personalized tools
PersonalAlign (Lyu et al., 2026)	✓	✓	do	×	static	Personalized GUI actions
LifeSim-Eval (Duan et al., 2026)	✓	×	passive	✓	interactive	Long-horizon implicit intention
Interactive and proactive-agent benchmarks						
Ask-before-Plan (Zhang et al., 2024)	×	×	ask	×	interactive	In-task constraint elicitation
PIRA-Bench (Chai et al., 2026)	✓	×	do	×	static	Proactive task recommendation
Pare-Bench (Nathani et al., 2026)	×	×	do	×	interactive	Proactive intervention
KnowU-Bench (Chen et al., 2026)	✓	✓	ask+do	×	interactive	Personalized + proactive mobile
ATRBench (ours)	✓	✓	**ask**	✓	**interactive**	**Cross-session preference acquisition**

Table 5: Comparison of ATRBench with representative personalization and proactive-agent benchmarks. Among the benchmarks we survey, ATRBench is the only one scoring proactive asking with later-task evaluation for the same user. ✓ supports; × not supported. **Proactive mode**: `passive` / `ask` / `do` (acts unsolicited) / `ask+do`.

pliedRate by 1.7–28.6 points (to 70.0–87.2 %). We report this as an observation rather than a measured effect, since each cell runs a single trial.

Appendix G.3 gives domain and action-surface breakdowns, Appendix H episode and LS/TS traces, and Table 16 the per-persona summary.

5 Related Work

Unlike existing personalization benchmarks that evaluate how disclosed preferences are applied, and proactive-interaction benchmarks that evaluate single-task capability, ATRBench evaluates cross-session preference acquisition (Table 5).

Personalized agents. On memory substrates (Packer et al., 2023), benchmarks test whether agents act on user-specific information: PrefEval over persistent memories (Zhao et al., 2025); MPT and LifeSim for personalized tool calls, with MPT modelling cross-session latent transfer (Yoon et al., 2026; Duan et al., 2026); and PersonalAlign for GUI agents (Lyu et al., 2026). In all of these the relevant user information is supplied as input or inferred from past interaction; ATRBench instead leaves it absent, obtainable only by asking.

Proactive and interactive agents. Interactive tool-agent benchmarks such as τ-bench test whether an agent can complete current tasks through tool use and user interaction (Yao et al., 2025). A complementary line gives the agent initiative: Ask-before-Plan elicits trip constraints during planning, scored inside the same plan (Zhang et al., 2024); PIRA-Bench and Pare-Bench score proactive assistance from logs and workflows (Chai et al., 2026; Nathani et al., 2026); and KnowU-Bench adds mobile personalization (Chen et al., 2026). In these the answer is consumed in the current task; ATRBench defers payoff to a later session. Closest to our setting, π-Bench also evaluates cross-session proactive assistance with delayed payoff (Zhang et al., 2026), but its proactivity is anticipating and acting on unstated intent, whereas ATR isolates explicit *asking* as the acquisition channel and binds each later task to a specific hidden rule, separating acquisition from application.

Preference elicitation. A methodological line studies optimal query selection: classical PE as a POMDP over utility functions (Boutilier, 2002) and LLM-based PE maximizing expected information gain over LLM-induced beliefs (Choudhury et al., 2025); Lin et al. (2022) recover linear rewards from *user-initiated* language. These assume a structured target and largely structured query–response spaces, whereas ATR's queries, answers, and tasks are open-ended natural language, so they do not directly transfer.

6 Conclusion

We identified the proactivity gap in long-lived LLM agents: the failure to acquire reusable user information before it is needed. We cast this as Ask-to-Remember (ATR), the task of deciding when and what to ask for later use, and built ATRBench to measure it while isolating asking from application. Across eight frontier LLM agents, oracle access to the missing rules raises future-task accuracy by 62.1–76.6 points, yet both prompting and forced asking leave acquisition sparse. We release ATRBench as a diagnostic testbed and next aim to ground its interaction contexts in real cross-session user trajectories.

Limitations

Synthetic-source rules. Personas are drawn from synthetic demographic-narrative records, and the standing rules, learning sessions, and rule-bound test sessions in ATRBench are produced by controlled LLM pipelines conditioned on each persona. The residual cohort summarized in Table 2 is therefore a surrogate, not a sample of any real user population. The construction is not bias-free, and the acquisition-side metrics (RuleAsk, RuleCov) are nontrivially shaped by the evaluated agent's own preference-category prior. The headline TSAcc gap is anchored by the oracle variant, which receives each rule answer directly and is insensitive to this prior dependence.

Future-task surrogate. For each persona, the set of rule-bound test sessions is a finite, controlled surrogate for the long tail of future tasks the same user might later issue. Absolute TSAcc therefore characterizes performance on this surrogate, not on any specific deployment cohort whose domain mix or rule density differs from ATRBench's.

Ethical Considerations

Potential risks. ATRBench is intended as a diagnostic benchmark for studying when agents should proactively ask for reusable user preferences. A system optimized for proactive acquisition without appropriate constraints could over-elicit personal information, ask intrusive questions, or retain preferences beyond the user's expectations. This risk is especially relevant if ATR-style objectives are transferred directly from our synthetic benchmark to deployed agents. Our benchmark framing treats asking as useful only when the requested information is reusable and consequential for future task execution, and we recommend that deployed systems pair proactive acquisition with explicit user control over what is remembered, updated, and deleted.

References

Anthropic. 2026. Introducing Claude Opus 4.7. https://www.anthropic.com/news/claude-opus-4-7. Official release page; system card listed at https://www.anthropic.com/system-cards, accessed May 2026.

Craig Boutilier. 2002. A POMDP formulation of preference elicitation problems. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence, July 28 - August 1, 2002, Edmonton, Alberta, Canada*, pages 239–246. AAAI Press / The MIT Press.

Seamus Brady. 2026. Springdrift: An auditable persistent runtime for LLM agents with case-based memory, normative safety, and ambient self-perception. *CoRR*, abs/2604.04660.

Yuxiang Chai, Shunye Tang, Han Xiao, Rui Liu, and Hongsheng Li. 2026. PIRA-Bench: A transition from reactive GUI agents to GUI-based proactive intent recommendation agents. *CoRR*, abs/2603.08013.

Tongbo Chen, Zhengxi Lu, Zhan Xu, Guocheng Shao, Shaohan Zhao, Fei Tang, Yong Du, Kaitao Song, Yizhou Liu, Yuchen Yan, Wenqi Zhang, Xu Tan, Weiming Lu, Jun Xiao, Yueting Zhuang, and Yongliang Shen. 2026. KnowU-Bench: Towards interactive, proactive, and personalized mobile agent evaluation. *CoRR*, abs/2604.08455.

Deepro Choudhury, Sinead Williamson, Adam Golinski, Ning Miao, Freddie Bickford Smith, Michael Kirchhof, Yizhe Zhang, and Tom Rainforth. 2025. BED-LLM: Intelligent information gathering with LLMs and bayesian experimental design. *CoRR*, abs/2508.21184.

DeepSeek-AI. 2026. DeepSeek-V4: Towards Highly Efficient Million-Token Context Intelligence. https://huggingface.co/deepseek-ai/DeepSeek-V4-Pro. Official model card and technical report for DeepSeek-V4-Pro and DeepSeek-V4-Flash, accessed May 2026.

Feiyu Duan, Xuanjing Huang, and Zhongyu Wei. 2026. LifeSim: Long-horizon user life simulator for personalized assistant evaluation. *CoRR*, abs/2603.12152.

Google DeepMind. 2026a. Gemini 3 Flash Preview. https://ai.google.dev/gemini-api/docs/models/gemini-3-flash-preview. Official Gemini API model documentation; Gemini 3 Flash model card published December 2025, accessed May 2026.

Google DeepMind. 2026b. Gemini 3.1 Pro Preview. https://ai.google.dev/gemini-api/docs/models/gemini-3.1-pro-preview. Official Gemini API model documentation; model card at https://deepmind.google/models/model-cards/gemini-3-1-pro/, accessed May 2026.

Jessy Lin, Daniel Fried, Dan Klein, and Anca D. Dragan. 2022. Inferring rewards from language in context. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, pages 8546–8560. Association for Computational Linguistics.

Yibo Lyu, Gongwei Chen, Rui Shao, Weili Guan, and Liqiang Nie. 2026. PersonalAlign: Hierarchical implicit intent alignment for personalized GUI agent with long-term user-centric records. *CoRR*, abs/2601.09636.

Adyasha Maharana, Dong-Ho Lee, Sergey Tulyakov, Mohit Bansal, Francesco Barbieri, and Yuwei Fang. 2024. Evaluating very long-term conversational memory of LLM agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pages 13851–13870. Association for Computational Linguistics.

Carlos Martin, Craig Boutilier, Ofer Meshi, and Tuomas Sandholm. 2024. Model-free preference elicitation. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI 2024, Jeju, South Korea, August 3-9, 2024*, pages 3493–3503. ijcai.org.

MiniMax. 2026. MiniMax M2.7: Early Echoes of Self-Evolution. https://www.minimax.io/news/minimax-m27-en. Official release page; no separate formal technical report found, accessed May 2026.

Deepak Nathani, Cheng Zhang, Chang Huan, Jiaming Shan, Yinfei Yang, Alkesh Patel, Zhe Gan, William Yang Wang, Michael Saxon, and Xin Eric Wang. 2026. Proactive agent research environment: Simulating active users to evaluate proactive assistants. *CoRR*, abs/2604.00842.

NVIDIA Corporation. 2025. Nemotron-Personas-USA: Synthetic personas aligned to real-world distributions. https://huggingface.co/datasets/nvidia/Nemotron-Personas-USA. Hugging Face dataset.

OpenAI. 2026. GPT-5.4 Model. https://developers.openai.com/api/docs/models/gpt-5.4. Official API model documentation, accessed May 2026.

Charles Packer, Vivian Fang, Shishir G. Patil, Kevin Lin, Sarah Wooders, and Joseph E. Gonzalez. 2023. MemGPT: Towards LLMs as operating systems. *CoRR*, abs/2310.08560.

Joon Sung Park, Joseph C. O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. 2023. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology, UIST 2023, San Francisco, CA, USA, 29 October 2023-1 November 2023*, pages 2:1–2:22. ACM.

Qwen Cloud. 2026. Qwen3.6-Plus. https://www.qwencloud.com/models/qwen3.6-plus. Official model documentation; no exact public technical report found, accessed May 2026.

Alireza Salemi, Sheshera Mysore, Michael Bendersky, and Hamed Zamani. 2024. LaMP: When large language models meet personalization. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pages 7370–7392. Association for Computational Linguistics.

Jingjing Xu, Yuechen Wang, Duyu Tang, Nan Duan, Pengcheng Yang, Qi Zeng, Ming Zhou, and Xu Sun. 2019. Asking clarification questions in knowledge-based question answering. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*, pages 1618–1629. Association for Computational Linguistics.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik R. Narasimhan. 2025. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

Yejin Yoon, Minseo Kim, and Taeuk Kim. 2026. Latent preference modeling for cross-session personalized tool calling. *CoRR*, abs/2604.17886.

Haoran Zhang, Luxin Xu, Zhilin Wang, Runquan Gui, Shunkai Zhang, Haodi Lei, Zihao He, Bingsu He, Chicheng Qin, Tong Zhu, Xiaoye Qu, Yang Yang, Yu Cheng, and Yafu Li. 2026. π-Bench: Evaluating proactive personal assistant agents in long-horizon workflows. *Preprint*, arXiv:2605.14678. ArXiv preprint.

Xuan Zhang, Yang Deng, Zifeng Ren, See-Kiong Ng, and Tat-Seng Chua. 2024. Ask-before-plan: Proactive language agents for real-world planning. In *Findings of the Association for Computational Linguistics: EMNLP 2024, Miami, Florida, USA, November 12-16, 2024*, Findings of ACL, pages 10836–10863. Association for Computational Linguistics.

Siyan Zhao, Mingyi Hong, Yang Liu, Devamanyu Hazarika, and Kaixiang Lin. 2025. Do LLMs recognize your preferences? evaluating personalized preference following in LLMs. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. 2024. MemoryBank: Enhancing large language models with long-term memory. In *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2014, February 20-27, 2024, Vancouver, Canada*, pages 19724–19731. AAAI Press.

A Benchmark Construction

This appendix records the implementation details needed to reproduce and audit ATRBench. Sections A–I cover, in order, dataset construction, runtime framework, scaffold calibration and validation, evaluation protocol, detailed experimental settings, compute budget, additional results, worked trajectory examples, and prompt listings. The main paper states the task and construction principles; the appendix expands the mechanical choices behind them.

A.1 Persona Cohort and Ingest

ATRBench draws 20 personas from Nemotron-Personas-USA (NVIDIA Corporation, 2025) through a pre-selection pass over a larger candidate pool. Each candidate first completes Stage A rule generation and two-axis QC (§A.3); only candidates whose QC-kept rule count falls in $[12, 20]$ are admitted. Among admitted candidates we balance for rule-domain coverage: every selected persona spans 3–6 domains, the cohort covers all six supported domains, and no single domain exceeds 35% of cohort rules. Selection uses only rule-stage signals and persona demographics; no downstream TS/LS or model-behavior results enter the cohort decision. The retained cohort holds 12–20 rules per persona (mean 17, median 17.5; 340 retained rules total). The main empirical sweep runs end-to-end on all 20 personas; Table 8 lists the retained test-session and learning-session counts.

The Nemotron-Personas-USA dataset card reports the dataset under the Creative Commons Attribution 4.0 International license (CC BY 4.0). We cite NVIDIA Corporation as the data creator and use the dataset as a source artifact for constructing ATRBench personas.

PII and offensive-content checks. ATRBench does not collect real user interactions. The source personas are synthetic records, and the standing rules, learning sessions, test sessions, local references, and tool states are generated artifacts. During construction, we check the retained cohort and public examples for direct contact or identity identifiers (e.g., email addresses, phone numbers, physical addresses, account or payment credentials, and government identifiers) and for overtly offensive content; candidates containing such material are regenerated or dropped. Runtime identity slots such as `contact_info`, `shipping_address`, `payment_method`, `traveler_info`, and `guest_info` are synthetic environment fields and are not exposed as user-provided task parameters. Released persona labels are pseudonymous synthetic identifiers rather than real user identities.

For each persona, we keep two views. The full structured Nemotron record (demographics plus narrative subfields) grounds rule generation and learning-session generation. A separate LLM pass compresses the narrative subfields into a $\leq$500-word background card used by the user simulator at runtime. Rule-bound test-session generation receives only the rule, not the persona.

Domain	Domain tools
commerce	13
reservation	13
travel	17
communication	9
scheduling	10
workspace	11
Base confirmation tool	1

Table 6: ATRBench tool ontology. The base tool is `get_user_confirmation`, inherited by every domain toolkit and used for confirmation-chain rules.

A.2 Tool Ontology and Environment

The executable environment exposes six personal-assistant domains. Each session exposes the full tool surface for its domain, while identifiers must be discovered through search, list, or track tools rather than copied from the prompt. Tool state is local to a session, and cross-session carry-over occurs only through the context layer described in §3.1. The domain action space contains 73 domain tools plus the base confirmation tool, giving 74 tools used for rule binding and evaluation.

A.3 Standing-Rule Generation and Quality Control

Rule generation receives the structured persona and the bindable tool ontology, and produces $N = 24$ candidate rules per persona in a single GPT-5.4 call. Each candidate carries a third-person rule statement, a first-person canonical answer, a counterfactual default, supporting evidence, a `check_type`, and an `action_step`. The counterfactual default is retained on the kept rules because the downstream test-session generator uses it to design reference decoys that pull a rule-blind agent toward the wrong choice (§A.4). A deterministic validator then checks field presence, legal `check_type`, legal tool and parameter names, and compatibility

Type	Binding semantics
`tool_identity`	The rule specifies which tool should be called.
`param_id`	The rule specifies which referenced object should be selected through an `*_id` or `*_ids` argument.
`param_enum`	The rule specifies a closed enum value on the selected tool.
`confirm`	The rule requires `get_user_confirmation` before a bound mutate tool.

Table 7: The four action-surface tags (`check_type` values) used by rule generation, test-session construction, and trajectory matching.

between the action-surface tag and the ontology.

A subsequent GPT-5.4 call scores each surviving candidate on two independent axes. `counter_default` asks whether a rule-blind helpful agent would already take the gold action; `binding_sound` asks whether the rule naturally maps to the proposed tool-level binding. Each axis is rendered with a curated few-shot pool of positive and negative cases, and a rule is retained only when both axes return `yes`. Across the 20-persona ATRBench dataset, 496 candidates are produced and 340 are retained (68.5% pass rate; mean 17.0 rules per persona, range 12–20). The four action-surface tags (recorded in the rule's `check_type` field) used downstream are summarized in Table 7.

A.4 Rule-Bound Test-Session Generation

For each retained rule, a single GPT-5.4 call generates a candidate test session conditioned on the rule, the domain tool specification, the reference schema, and few-shot exemplars. The generator follows a five-step internal procedure: decompose the rule's discriminative dimension; using the rule's counterfactual default, design reference objects whose decoys pull a rule-blind agent toward the wrong choice; triple-filter gold against rule, instruction, and decoy disadvantage; write a self-contained instruction free of rule-direction wording and the gold reference identifier; self-check. The output carries an instruction, a gold value, and local reference objects; the executable success label is then derived programmatically from the rule binding and the gold value, rather than accepted from the generator.

Each candidate passes through a static deterministic check (schema compliance, legal tool arguments, reachable reference objects, environment constructibility) and an LLM-judge ensemble of three independent Gemini 3 Flash Preview samples per bound. The upper bound runs each sample with the rule's canonical answer injected as user context; the lower bound runs the same prompt without it. Each bound matches when at least two of the three samples reach the gold action, and a candidate passes overall when the upper bound matches and the lower bound does not. Failed candidates are refined against the QC failure trace for up to two rounds and dropped if still invalid. Across the 20-persona ATRBench dataset, 340 candidates are screened and 284 are retained (83.5% pass rate); rules whose candidates ultimately fail are excluded, so retained rules and test sessions form a one-to-one pairing.

A.5 Learning-Session Generation and Sequence Assembly

Learning-session construction has two GPT-5.4 stages. A skeleton call per persona produces a pool of $3|\mathcal{T}_u| + 2$ thin themes (clamped to $[8,\ 100]$) that are temporally ordered across the persona's six tool domains, each carrying only a domain tag and a one-line theme. A fill call per skeleton then instantiates the theme into a complete learning session with a vague user intent, concrete task parameters, local references, and an oracle tool trajectory. Each fill is validated by a deterministic shape check and a dry-run of the gold trajectory against a live environment instance, refined for up to one round on failure, and dropped otherwise. Across the 20-persona ATRBench dataset, 880 of 894 skeletons (98.4%) yield a valid session.

The final sequence retains $2|\mathcal{T}_u|$ sessions, greedily selected from the pool. We first add every session whose oracle trajectory touches a decision-surface tool used by some retained test rule, then fill remaining slots from sessions whose tools overlap no retained rule. When the signal pool already exceeds the target length, a random subset of size $2|\mathcal{T}_u|$ is kept. Selected sessions are emitted in their generated temporal order. In the final 20-persona cohort, $205/284 = 72.2\%$ of the retained standing rules have at least one selected learning session whose oracle trajectory touches the rule's action surface, with a per-persona range of 55–91%.

A.6 Dataset Statistics

Table 8 reports per-persona dataset statistics for the 20-persona evaluation cohort. TS and LS count retained rule-bound test sessions and final learning-sequence sessions, respectively; Domains counts

Persona	TS	LS	Domains
anna_strahan	13	26	6
cassandra_tovar	11	22	5
cecilia_lisette_garciaperez	12	24	6
charlie_james	18	36	6
eduardo_rivera	17	34	6
esther_hoitsma	14	28	6
gabriel_campbell	15	30	6
iris_parker	12	24	5
jennifer_philmon	12	24	5
john_brown	12	24	6
joseph_eldridge	17	34	6
maximo_esteban_irizarry	11	22	6
michael_argo	17	34	6
nancy_marr	14	28	5
rafael_ortiz	11	22	6
romisha_shields	14	28	6
roosevelt_brown	15	30	6
samantha_bello	15	30	6
theodora_tong	19	38	6
vanessa_davis	15	30	6

Table 8: Per-persona dataset statistics for the 20-persona evaluation cohort.

the tool domains represented by the retained test sessions.

B Runtime Framework

B.1 Agent Output Types

The agent uses native function calling and emits, on each turn, an assistant payload that may contain tool calls and/or free-form content.

Control-plane tools. On top of the environment's domain tools (Appendix A.2), the orchestrator prepends one phase-specific synthetic tool: `send_to_user(output, reason)` in LS only (the agent's sole user-facing channel) and `finish_session()` in TS only (the agent's terminator). Domain tools execute against the live environment; the synthetic tools are intercepted by the orchestrator and never reach the environment.

Outputs in a learning session. On each LS turn the agent emits either (i) zero or more domain tool calls, (ii) zero or one `send_to_user(output, reason)` call, or (iii) free-form text without any tool call. The first two types may co-occur in a single assistant turn; free-form text alongside tool calls is treated as internal narration under standard tool-calling protocol and is not user-visible. Free-form text without any tool call (a *text turn*) is off-protocol: the orchestrator intercepts it and injects a `<scaffolding_note>` (Appendix B.4) into the agent's next turn asking it to re-emit through `send_to_user`. If the agent emits more than one `send_to_user` call in the same turn, the orchestrator keeps the first and drops the rest to preserve the tool-call–tool-response pairing invariant; when `send_to_user` co-occurs with other tool calls, all tools still execute in declared order. Both situations are recorded as diagnostic events but are not blocked.

The send_to_user contract. `output` carries the text delivered to the user simulator and is the only agent string that enters the visible transcript. `reason` is an internal intent string consumed by the Router as auxiliary evidence; it is withheld from the simulator, Classifier, evaluator, and cross-session context. Routing every user-visible utterance through this single channel lets the Router inspect all and only the messages that could plausibly be standing-rule asks, without disturbing the conversation.

Outputs in a test session. Test sessions are user-offline and have no `send_to_user` tool. The agent emits domain tool calls and may include plain text alongside them; plain text is allowed and not intercepted. The session ends when the agent calls `finish_session()` or hits the maximum step or wall-clock limit.

B.2 Router and Classifier Scaffold

Router. The Router decides the speech-act type of a `send_to_user` turn. Its input is the agent's user-visible `output` and internal `reason` for the current turn; it returns a binary verdict `is_strict_rule_question` together with, on positives, a verbatim `rule_question_span` copied from `output`. Direct questions and soft or indirect asks both count when they ask about a future or default behavior; recall of past rules, status updates, and quoted draft content are excluded. The implementation calls a fixed Gemini 3 Flash Preview backend with a versioned few-shot bank. Framework faults (LLM transport exhaustion, a malformed schema, or a positive verdict without a span) cell-abort the run rather than silently degrading to the task route, so reported metrics never reflect missing speech-act decisions.

Classifier. The Classifier matches a Router-positive question span against the episode's candidate rules. Its input is the span and the rule pool, so its attention is on topic-to-rule matching rather than intent classification. A rule fits only when both the rule's trigger context covers the situation in the

question and the rule's preferred behavior directly answers the action, value, or choice the question proposes; if either condition is weak or unclear, the Classifier returns `null`. The implementation calls a fixed GPT-5.4 backend with a versioned few-shot bank. An unknown rule identifier (one not in the pool) is retried once and then cell-aborts; a non-dict result is treated as a framework fault and cell-aborts immediately without retry.

Both modules' calibration sets, few-shot banks, and held-out validation accuracy are reported in Appendix C.

B.3 User Simulator Mechanism

The user simulator π_u plays the user during learning sessions only; test sessions are user-offline. By design π_u holds only the current task parameters and the persona narrative; it has no inner long-term preferences and no awareness of the rule-acquisition channel beyond a transient per-turn hook. The implementation calls a fixed `gpt-5.4` for both opening and reply turns.

Opening. The opening call sees only `reason_for_call` (an abstract user intent) plus a sanitized hint derived from the first oracle step, limited to a per-tool allowlist of safe argument keys. The persona narrative, full task parameters, references, and the rest of the oracle trajectory are intentionally withheld at this stage so task-specific facts surface only when the agent asks. An empty reply is retried twice; persistent empty replies cell-abort the run.

Replies. Subsequent calls see the persona narrative, full task parameters, local references, `reason_for_call`, and the oracle trajectory. π_u paraphrases task parameters naturally rather than parroting raw values, and does not pick sides on closed-form choices the agent presents without an explicit task answer. Role reversal is used internally to the GPT-5.4 call (agent text occupies the user role, π_u's replies occupy the assistant role); this is invisible to the agent and the rest of the system.

Hook integration. On Router-positive turns the orchestrator appends a transient `<rule_answer>` directive as a separate user-role message instructing π_u to emit a `<RULE_ANSWER>` token once in its reply at the natural answer position. π_u sees only the directive; the orchestrator substitutes `<RULE_ANSWER>` after the reply with the matched canonical answer (Classifier hit) or with the fixed deflect phrase (Classifier miss/error). If π_u drops the token, the orchestrator retries up to twice, uniformly across Classifier hit, miss, and error, and appends a guaranteed-delivery fallback if the token is still absent. The hook is not persisted to the trajectory.

Termination. π_u has one control-plane tool, `mark_task_complete()`, which signals intent end: the orchestrator stamps a `USER_END` marker on the user message and ends the learning session. Any free-text emitted alongside the tool call is discarded. π_u does not gate task success on its own: the agent's stated completion is taken at face value, and the evaluator is the final arbiter.

B.4 Runtime Orchestration

The runtime orchestrator handles every transition in a learning-session turn and decides what the next actor sees. Test sessions are user-offline and terminate when the agent calls `finish_session()` or hits a max-step or wall-clock limit.

Output dispatch. Domain tool calls execute against the environment and their results are appended to the agent's next-turn context, with no scaffold processing. A `send_to_user` call is forwarded to the Router–Classifier scaffold (Appendix B.2). A text turn (assistant turn with no tool calls and non-empty content) triggers an inner retry loop of up to three attempts, each appending a transient `<scaffolding_note>` to the LLM input asking the agent to re-emit through `send_to_user`; on a successful rescue, the original off-protocol message is popped from the trajectory and the rescued response takes its slot. If all retries return text turns, the orchestrator falls back to routing the original leak text through Router and Classifier as if it had come from `send_to_user`.

Adjudication and hook injection. On every `send_to_user` or rescue-exhausted leak, the Router is invoked over `(reason, output)`. On a positive verdict, the Classifier is invoked on the extracted span against the persona's rule pool. The orchestrator then appends a transient `<rule_answer>` directive to the user simulator's next input, referencing the Router's verbatim span and instructing the simulator to emit a `<RULE_ANSWER>` token. The directive is uniform across Classifier hit, miss, and error; the simulator is Classifier-blind. On a Router-negative turn, no directive is injected and the simulator answers normally.

Placeholder substitution. After the simulator replies, the orchestrator substitutes <RULE_ANSWER> with the matched canonical answer on a Classifier hit, or with the fixed deflect phrase `no strong preference there - your call` on a Classifier miss or error. If the simulator drops the token, the orchestrator retries the simulator up to twice, uniformly across hit, miss, and error, and appends the replacement at the end as a guaranteed-delivery fallback if the token is still absent. End-intent is normalized asymmetrically: on a hit, an `end` intent is overridden to `continue` so the answer is delivered before the session closes; on a miss or error, an `end` intent is respected. When the simulator calls `mark_task_complete`, the orchestrator stamps a ###USER_END### marker on the user message and ends the learning session.

Cross-session context. After each learning session ends, the runtime renders the cleaned session transcript and appends it to the cross-session context block embedded in the next session's system prompt. The renderer preserves user-visible dialogue and business tool evidence as `[user]: ...`, `[assistant to user]: ...`, `[tool call]: name(args)`, and `[tool result]: ...` in chronological order; `send_to_user.reason`, the synthetic acknowledgments for `send_to_user` and `finish_session`, scaffolding notes, and assistant content on tool-call turns are hidden as internal protocol. Real learning sessions render as numbered `[Session` i`]` blocks; the oracle variant skips learning and instead renders `[Prior statement` i`]` blocks containing first-person canonical answers.

Logged events. The orchestrator records protocol events per turn, including `send_event` (with the visible `output` and internal `reason`), `route_decision` (Router verdict and span), `cls_verdict` (matched rule id or null), `off_protocol_ask` (a text turn that was not rescued, with the leak text), `hook_appended` (one per retry attempt), `stu_mixed_with_tools` (a `send_to_user` sharing a turn with other tool calls), `stu_duplicate_same_turn` (a `send_to_user` duplicate dropped to preserve tool pairing), and `rule_hook_token_missing` (the simulator failed to emit <RULE_ANSWER> despite the directive). These events are used for diagnostics and do not create test-time access to the hidden rules.

B.5 Diagnostic Variants and Prompt Blocks

Prompt block layout. The agent's system prompt is assembled from a fixed set of named blocks: a shared `base` (identity and ID-discovery rules), a per-domain `domain_<d>` block (ID discovery paths and tool shape constraints), the cross-session `<context>` block, a phase block (`mode_learning` or `mode_test`), and in learning sessions a universal `send_protocol` block that establishes `send_to_user` as the agent's sole user-facing channel. The phase and variant blocks are placed after `<context>` so protocol constraints stay close to the prompt tail where attention is strongest.

Variant configurations. Five canonical variants are exposed in the runtime; `default`, `atr`, and `always_ask` run a full learning + test episode, while `oracle_full` and `oracle_target` skip learning and run only the test phase. `default` sees no standing-rule block at all and measures spontaneous rule-asking. `atr` appends a shared `standing_rule_def` block (defining what counts as a standing rule) followed by a `variant_atr` block: a soft permission to ask about a standing rule of the user's that may prove useful later, with a cost–benefit reminder (patience cost vs. future-error reduction). `always_ask` appends the same `standing_rule_def` plus a `variant_always_ask` block requiring exactly one standing-rule question after the current task is largely complete. Enforcement is prompt-only: the orchestrator does not block `finish_session` when the agent disobeys, and a separate `forced_rule_ask_compliance` metric records adherence. The `variant_atr` and `variant_always_ask` blocks never mention `send_to_user` or `reason`, so the WHEN-to-ask nudge is decoupled from the HOW-to-ask channel.

Oracle variants. `oracle_target` injects only the current test session's target rule canonical answer; `oracle_full` injects all rule canonical answers for the persona. Both deliver the answers as synthetic prior user statements (rendered as `[Prior statement` i`]` blocks; Appendix B.4) through the same `<context>` channel that learning variants use, so the rule-presentation format is identical across variants. The main paper reports `oracle_target` as the application upper bound; `oracle_full` is included as a broader-context reference.

Excluded variants. A `naive` variant (no learning, no rule injection) is reserved for follow-up

work but is excluded from the runtime in the present launch.

C Scaffold Calibration and Validation

The Router and Classifier (Appendix B.2) are calibrated outside the benchmark episodes from manually reviewed bad cases observed in experiment traces. These examples are not rule-bound test sessions and do not enter TSAcc; they are used only to stabilize the speech-act and rule-matching scaffold. We describe each module's calibration set, few-shot bank, and validation accuracy separately below.

C.1 Router Calibration

Calibration set. The Router calibration set targets two decisions. The binary set contains 11 manually labeled rows harvested from phase-2 Router bad cases, with 3 positive and 8 negative examples. The span set contains 104 manually positive rows and checks whether the Router copies the complete standing-rule ask sentence as a verbatim substring of the agent output. The negative cases distinguish current-task choices, artifact-formatting preferences, and declarative memory offers from true standing-rule questions; the positive span cases fix boundary conventions, such as keeping lead-ins like "for future" or "since we are looking at appointments" when they are part of the same ask sentence.

Few-shot bank. The active Router bank contains 22 pattern-level examples: 14 positives and 8 negatives. It was derived from Router bad-case categories and teaches four recurring boundaries: current-task preference questions, declarative memory offers, soft or indirect standing-rule asks, and exact span extraction with sentence-level lead-ins preserved.

Final-sweep validation sample. After freezing the Router prompt and few-shot bank, we ran a validation-only manual audit on the completed 20-persona non-oracle learning-session sweep. The population contains 47,119 Router decisions: 4,838 Router-positive turns and 42,281 Router-negative turns. We sampled 100 rows from each predicted polarity bucket, using simple random sampling within each bucket. The sample is not stratified by model, variant, or persona. The audit unit is the paired `send_to_user` event and same-turn Router decision; oracle cells, test-session trajectories, and incomplete cells are excluded. The sampled rows were not used to tune prompts or few-shot examples.

Bucket	Rows	TP exact	Span mm.	FP	TN/FN
Router-positive	100	98	2	0	–
Router-negative	100	–	–	–	100/0
Total	**200**	**98**	**2**	**0**	**100/0**

Table 9: Manual validation of the frozen Router on a balanced 200-row sample from the final 20-persona sweep. "Span mm." denotes a gold-positive Router-positive row where the binary decision was correct but the extracted span was not an exact verbatim match.

Manual review protocol. A row is gold-positive only when the visible assistant output both expresses a future/default/standing-rule intent and actually asks the user about that long-term preference or rule. Private `reason` text is used only as provenance and cannot turn an invisible intent into a positive label. Gold-negative cases include current-task choices, current-artifact formatting choices, declarative memory offers, acknowledgements of already-applied defaults, quoted draft text, status updates, and generic follow-ups. For gold-positive rows, the reviewed span is one complete verbatim standing-rule ask sentence, preserving same-sentence future/default lead-ins. The review was single-pass; we do not report second-annotator overlap.

Validation accuracy. The Router achieved 100/100 = 100.0% binary precision on predicted standing-rule questions. Among manually positive Router-positive rows, exact span accuracy was 98/100 = 98.0%. The Router-negative false-negative rate was 0/100 = 0.0%. On this balanced paper-facing sample, binary accuracy was 200/200 = 100.0%; requiring both the binary verdict and exact span to be correct gives a span-sensitive accuracy of 198/200 = 99.0%. The two disagreements were both span mismatches caused by omitted markdown bold delimiters inside otherwise correct standing-rule asks.

C.2 Classifier Calibration

Calibration set. The Classifier calibration set contains 271 manually labeled Router-positive questions paired with persona rule pools. The current file has 34 should-hit rows and 237 should-miss rows, reflecting the main observed failure mode: over-matching a plausible recurring question to a hidden rule whose trigger or behavior does

not actually answer that question. Each row records the original agent text, the Router-extracted question span, the candidate rule pool, the gold rule id or null, provenance, and a manual note.

Few-shot bank. The active Classifier bank contains 68 examples: 39 hits and 29 misses. It extends earlier calibration banks with curated broad-standing-rule hit cases and sharper communication and ground-transport boundaries.

Final-sweep validation sample. After freezing the Classifier prompt and few-shot bank, we ran a validation-only manual audit on the completed 20-persona non-oracle learning-session sweep. The population contains 4,838 Classifier verdicts, all triggered by Router-positive turns: 479 predicted hits and 4,359 predicted misses. We sampled 100 rows from each predicted polarity bucket, using simple random sampling within each bucket. The sample is not stratified by model, variant, or persona. The sampled rows were not used to tune prompts or few-shot examples.

Manual review protocol. The reviewed unit is the Router-extracted `rule_question_span` paired with the episode rule pool. A gold hit requires the rule's trigger context to cover the question and the rule's preferred behavior to answer the action, value, or choice proposed by the question. Broad standing-rule asks are positive when the question names a rule field that one pool rule naturally answers; similarly, a broader rule can cover a narrower operational question when applying the rule answers the requested default, such as project-first file organization, draft-first messaging, or event-relative reminders. Same-domain overlap, tool overlap, adjacent workflow preferences, content style, tracking defaults, implementation details, and broader or narrower recipient scope are not sufficient. Same-turn assistant output is used only to resolve local references such as "like this"; private `reason` text is not used to create a match. The review was single-pass; we do not report second-annotator overlap.

Validation accuracy. The Classifier achieved 90/100 = 90.0% precision on predicted hits. The predicted-miss false-negative rate was 4/100 = 4.0%. On this balanced paper-facing sample, exact rule-id accuracy was 186/200 = 93.0%. Among manually positive rows, recall was 90/94 = 95.7%, and the corresponding F1 was 92.8%. The main disagreement pattern was over-matching adjacent but different decision fields, including search or filtering strategies, output-format details, scope mismatches, and broad booking questions that span beyond a single rule field. The four false negatives were broad-but-answerable community drafting, message-labeling, and restaurant asks.

Bucket	Rows	TP	TN	FP miss	FN
cls-positive	100	90	–	10	–
cls-negative	100	–	96	–	4
Total	**200**	**90**	**96**	**10**	**4**

Table 10: Manual validation of the frozen Classifier on a balanced 200-row sample from the final 20-persona sweep. "FP miss" denotes a predicted rule id where manual review found that no rule in the pool covered and answered the question. No wrong-rule false positives occurred in this sample.

D Evaluation Protocol

D.1 Evaluation and Aggregation

The main evaluation uses raw transcript context. Learning sessions are run serially so that the context block grows over time; after the final learning session, the context is frozen and shared across test sessions without write-back. Test sessions are user-offline and terminate when the agent calls `finish_session()` or hits a max-step or wall-clock limit. All primary metrics are computed at the persona level and macro-averaged across personas: RuleAsk (Router-positive sends per learning session), AcqPrec (share of strict rule questions that the Classifier maps to a hidden rule), RuleCov (distinct Classifier-hit rules per retained rule pool), AppliedRate (TS pass rate over learnable rules that are Classifier-confirmed as acquired during learning), and TSAcc (TS pass rate over all retained rules). A learnable rule is one with at least one selected learning session whose oracle trajectory touches the rule's action surface; concretely, AppliedRate is `covered_hit_pass`/(`covered_hit_pass` + `covered_hit_fail`).

D.2 Action Matching

Each retained test session has one required action. The matcher groups tool calls by assistant turn, so parallel tool calls in a single assistant message are evaluated as one batch. It scans the trajectory until it finds the first batch containing the required tool; later batches are ignored once this batch decides

the verdict.

For `tool_identity` rules, any call to the required tool in that first matching batch is sufficient. For `param_id`, `param_enum`, and `confirm` rules, every call to the same required tool in the first matching batch must match the specified comparison arguments; this rejects same-turn shotgun behavior such as calling the same mutate tool with both correct and incorrect targets. Extra calls to different tools in the same batch are ignored. Scalar arguments use exact equality, list arguments use set equality, and `confirm` rules compare `target_params` by safe typed fields of the target tool.

E Detailed Experimental Settings

This section details the models evaluated as the agent under test, their inference configuration, and the sweep execution settings. Table 11 lists backend identifiers and exact official source pages. We cite the most specific public source available for each backend: official technical reports where public, otherwise official system cards, model cards, release pages, or API documentation; we do not treat broader family reports as exact technical reports for proprietary API models.

Model access terms. The evaluated frontier models are proprietary API services rather than redistributed model weights. We access them through their provider APIs under the corresponding provider terms, and Table 11 lists the official source page used for each model artifact.

Sweep and execution. The main sweep crosses the 8 evaluated agents, the 4 variants (Table 1), and the 20-persona cohort, giving $8 \times 4 \times 20 = 640$ (model, variant, persona) cells. Each cell runs one learning sequence followed by all rule-bound test sessions for that persona. Sessions are single-trial with a 20-turn cap, which serves as the max-step limit referenced in Appendix D.1; primary metrics are computed per persona and macro-averaged as defined there. The sweep runs at a cell-level concurrency of ten with intra-cell test-session concurrency of one; scaffold backends and shared infrastructure are as in §4.1.

Per-model reasoning configuration. For the agent under test, we enable each provider's reasoning or thinking mode where exposed, with the per-model settings below. GPT-5.4 uses `reasoning_effort=high`; Claude Opus 4.7 uses `output_config.effort=high` with adaptive thinking enabled; Gemini 3 Flash Preview and Gemini 3.1 Pro Preview use `thinkingLevel=high`; DeepSeek V4 Pro and Flash use `thinking={type:enabled, reasoning_effort:high}`; Qwen3.6-Plus uses the boolean toggle `enable_thinking=true`; MiniMax M2.7 exposes no reasoning toggle and always interleaves thinking. The user simulator, Router, and Classifier use each provider's default and do not enable reasoning.

Display name	Backend id	Src.
GPT-5.4	`gpt-5.4`	API docs
Claude Opus 4.7	`claude-opus-4-7`	rel.; card
Gemini 3 Flash Preview	`gemini-3-flash-preview`	API; card
Gemini 3.1 Pro Preview	`gemini-3.1-pro-preview`	API; card
Qwen3.6-Plus	`qwen3.6-plus`	API docs
MiniMax M2.7	`MiniMax-M2.7`	rel.; card
DeepSeek V4 Pro	`deepseek-v4-pro`	report
DeepSeek V4 Flash	`deepseek-v4-flash`	report

Table 11: Evaluated API model inventory. The main paper uses display names; exact backend identifiers and official provider pages are reported here for reproducibility.

F Compute Budget and Inference Cost

We report compute budget across the two LLM-using phases of the project, benchmark construction and the main 640-cell evaluation sweep, under two billing scenarios: a no-cache list price that bills every input token at the provider's standard input rate, and a with-cache cost that bills cached input tokens at the provider's cache-hit rate where supported. Aggregate phase-level numbers are in Table 12. Dollar figures use the per-million-token rates posted on the inference platform at the time of the sweep; cache-hit rates are read from per-cell token accounting written by the runner.

Benchmark construction. Producing the 20-persona dataset, including persona ingest, standing-rule generation and QC, test-session generation with iterative bound checks, and learning-session skeleton and fill, consumed about 29 M input and 15 M output tokens across about 5,200 LLM calls. Authoring calls use GPT-5.4 and dominate dollar cost; QC trace simulation uses Gemini 3 Flash

Phase / role	In (M)	Out (M)	Hit	$ list	$ cached
Benchmark construction	29	15.0	–	138	138
Sweep, agent under test	2,208	23.0	64 %	3,114	1,394
Sweep, scaffold roles	299	15.0	20 %	298	196
Total	**2,536**	**53.0**	60 %	**3,550**	**1,728**

Table 12: Aggregate compute budget across the benchmark-construction and main-sweep phases. In and Out are total input and output tokens in millions. Hit is the fraction of input tokens served from provider prompt cache, averaged across the constituent calls. $ list bills every input token at the provider's standard input rate; $ cached bills cached input tokens at the provider's cache-hit rate where supported. Construction uses one-shot calls and has no cache reuse.

Preview. Test-session generation alone contributes about two-thirds of the construction token budget because each candidate rule passes through three rounds of upper-bound and lower-bound checks. Total construction cost is about $140, or about $7 per persona; this phase issues one-shot calls without prompt-cache reuse.

Main sweep. The 640-cell sweep (8 models, 20 personas, 4 variants) executed 13,632 learning sessions and 9,088 test sessions against the agent under test, plus the corresponding calls to the user simulator, Router, and Classifier. Aggregate agent-under-test input volume is on the order of 2.2 B tokens with about 23 M output, the low output-to-input ratio reflecting that most cells are tool-call dominated. The user simulator and the scaffold are pinned to fixed backends (GPT-5.4 for the user simulator and the Classifier; Gemini 3 Flash Preview for the Router) and add a roughly constant 300 M input and 15 M output tokens across all eight evaluated agents, so the scaffold cost does not bias the per-model comparison.

Cache and total cost. Six of the eight evaluated agents and two of the three scaffold roles serve a measurable fraction of their input tokens from provider prompt cache, with measured cache-hit rates ranging from 36 % (Gemini 3.1 Pro Preview) to 92 % (DeepSeek V4 Flash); the Classifier hits cache at about 83 % because its prompt is short and reused across rule-routing calls. Across the full sweep, about 1.4 B of 2.2 B agent input tokens hit cache, a sweep-level hit rate of roughly 64 %. Aggregate project cost is about $3,550 at posted list price and about $1,730 after applying cache-hit discounts, with Claude Opus 4.7 and GPT-5.4 contributing about 60 % of the with-cache total. Wall-clock for the sweep was about 36 hours under a cell-level concurrency of ten and an intra-cell test-session concurrency of one.

G Additional Results

G.1 Run Completion and Health

All 640 planned cells in the 20-persona × 8-model × 4-variant sweep completed, with no missing learning or test sessions. We separate this completion check from runtime health: Table 13 audits normal versus abnormal session termination, and the paragraph below summarizes learning-session channel repair and scaffold integrity.

Variant	LS norm.	LS abn.	TS norm.	TS abn.
Default	4502/4544	42	2198/2272	74
ATR	4504/4544	40	2200/2272	72
Always	4510/4544	34	2203/2272	69
Oracle	–	–	2233/2272	39

Table 13: Runtime termination health over completed cells. LS normal denotes user-simulator `task_complete`; TS normal denotes agent `finish_session`. Abnormal terminations include max-step exits, no-progress exits, and empty-response exits. Oracle has no learning sequence.

The user-channel scaffold did not create missing cells or missing classifier decisions. Across learning variants, text-turn rescue recovered at least 99.0% of initially off-protocol user-facing outputs, duplicate `send_to_user` events occurred once, and Classifier runtime errors were zero. These channel-health diagnostics support the integrity of the acquisition metrics but are not used as evaluation outcomes.

G.2 Acquisition Diagnostics

Table 14 reports the full per-model acquisition diagnostics summarized in §4.3. Ask is RuleAsk, measuring how often the agent asks standing-rule questions during learning sessions; Cov is RuleCov, measuring how much of the persona's retained rule pool is acquired; App is AppliedRate, measuring whether learnable, classifier-confirmed acquired rules are later applied successfully; and GapRec is the fraction of the `default`-to-`oracle` gap recovered by the non-oracle variant.

Model	Ask↓	Cov↑	App↑	GapRec↑
ATR				
GPT-5.4	0.01	0.4	0.0	-2.2
Claude Opus 4.7	0.14	7.1	97.7	9.2
Gemini 3 Flash Preview	0.51	12.0	78.4	14.6
Gemini 3.1 Pro Preview	0.80	13.3	88.9	15.5
Qwen3.6-Plus	0.01	0.7	100.0	-1.2
MiniMax M2.7	0.00	0.0	–	1.3
DeepSeek V4 Pro	0.01	0.6	50.0	1.8
DeepSeek V4 Flash	0.05	2.1	100.0	-0.7
AVG ATR	**0.19**	**4.5**	**73.6**	**4.8**
Always				
GPT-5.4	1.00	12.7	80.9	13.5
Claude Opus 4.7	1.01	15.6	83.2	14.1
Gemini 3 Flash Preview	1.01	14.4	70.0	13.5
Gemini 3.1 Pro Preview	1.00	11.3	87.2	8.4
Qwen3.6-Plus	0.96	10.3	73.5	6.8
MiniMax M2.7	0.09	1.8	25.0	-3.6
DeepSeek V4 Pro	0.98	14.5	70.6	5.6
DeepSeek V4 Flash	0.97	13.3	71.4	4.0
AVG Always	**0.88**	**11.7**	**70.2**	**7.8**

Table 14: Full per-model acquisition diagnostics (20-persona macro). Ask = RuleAsk, measured per learning session; Cov = RuleCov; App = AppliedRate over learnable, classifier-confirmed acquired rules. Cov, App, and GapRec are reported in %.

G.3 Capability Breakdown

Aggregate TSAcc hides where the gap is concentrated. We therefore break Table 3 down by tool domain and by action-surface type.

Appendix Finding A1: the oracle gap is broad across domains and action surfaces. Figure 5 shows that `oracle` reaches 67.5–100 % across completed (model, domain) cells, while the three learning-sequence variants remain much lower. Commerce is hardest without rule knowledge: across `default`, `atr`, and `always_ask`, completed models stay between 1.9 and 15.1 % TSAcc. Communication and workspace are partially solvable even without acquisition, reaching 14.3–78.6 % and 30.3–63.6 %, respectively, which suggests that some domain decisions are recoverable from task-local cues or generic model priors. The benchmark therefore does not expose a single uniform failure; it separates domains where hidden standing rules dominate from domains where rule-naive behavior can sometimes succeed.

Figure 6 shows the same pattern by action surface. `oracle` reaches 74.6–97.6 % on `param_id`, `param_enum`, and `tool_identity`, confirming that all three action surfaces are executable when the

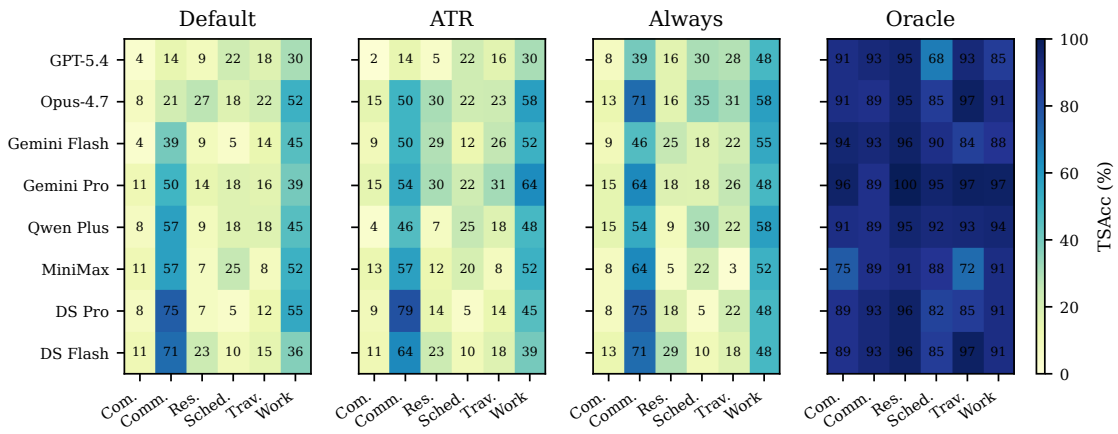

Figure 5: Domain-level TSAcc shows broad executable upper bounds but uneven non-oracle behavior. Each heatmap cell is TSAcc (%) aggregated over all 20 personas for one completed model, variant, and tool domain.

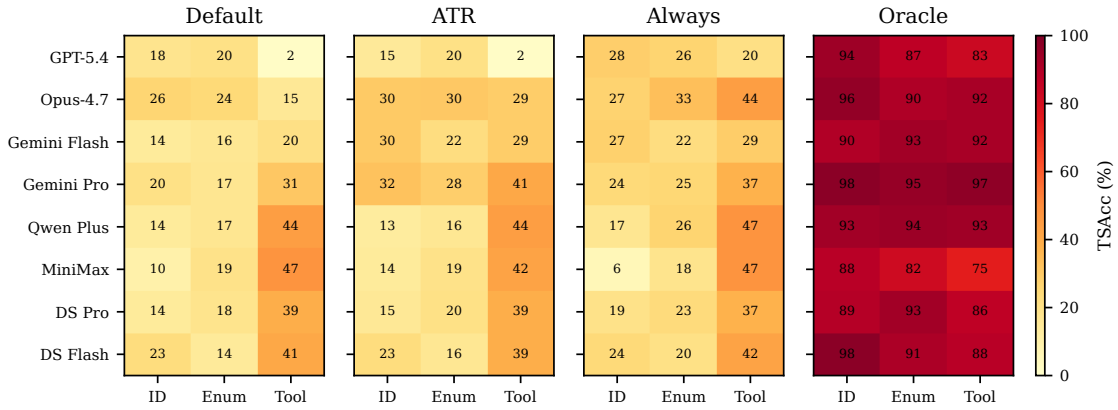

Figure 6: Action-surface breakdown for the three non-sparse check types. ID = `param_id`, Enum = `param_enum`, and Tool = `tool_identity`; confirmation rules are omitted from this figure because Table 2 shows that bucket is too small for stable accuracy claims.

rule answer is known. In contrast, non-oracle cells stay below 33 % for `param_id` and at or below 33.3 % for `param_enum`, while `tool_identity` is higher but still capped at 47.5 %. Future ATR methods therefore need both better elicitation and better binding of acquired natural-language rules to concrete tool arguments.

Numeric summaries. Table 15 provides compact numeric summaries for the heatmaps in Figures 5 and 6. Each entry is TS-micro accuracy over completed cells, pooling all personas and evaluated models for the corresponding variant and bucket. Table 16 reports the complementary persona-macro view, averaging across the eight evaluated models for each persona.

By domain

Variant	Com.	Comm.	Res.	Sched.	Trav.	Work
Default	8.0	48.2	13.2	15.0	15.2	44.3
ATR	9.9	51.8	19.0	17.5	19.1	48.5
Always	11.1	60.7	17.0	20.9	21.3	51.9
Oracle	89.4	91.1	95.5	85.6	89.9	90.9

By action surface

Variant	ID	Enum	Tool	Confirm
Default	17.1	18.3	29.9	0.0
ATR	21.4	21.6	33.1	6.2
Always	21.6	24.4	38.1	12.5
Oracle	92.9	90.5	88.1	12.5

Table 15: Capability TSAcc summaries (%, TS-micro). The upper block groups by domain and the lower block groups by action-surface type. The confirm bucket has only two retained rules.

Persona	Def.	ATR	Always	Oracle	Gap	Persona	Def.	ATR	Always	Oracle	Gap
anna_strahan	17.3	20.2	22.1	97.1	79.8	joseph_eldridge	33.1	31.6	39.7	89.7	56.6
cassandra_tovar	9.1	9.1	13.6	84.1	75.0	maximo_esteban_irizarry	12.5	15.9	17.0	92.0	79.5
cecilia_lisette_garciaperez	15.6	17.7	19.8	92.7	77.1	michael_argo	28.7	31.6	33.1	91.2	62.5
charlie_james	14.6	19.4	20.1	81.2	66.7	nancy_marr	15.2	24.1	21.4	86.6	71.4
eduardo_rivera	20.6	25.0	22.1	93.4	72.8	rafael_ortiz	17.0	22.7	22.7	98.9	81.8
esther_hoitsma	9.8	16.1	17.0	85.7	75.9	romisha_shields	25.9	30.4	37.5	84.8	58.9
gabriel_campbell	24.2	26.7	29.2	90.0	65.8	roosevelt_brown	15.8	20.8	27.5	98.3	82.5
iris_parker	18.8	20.8	20.8	92.7	74.0	samantha_bello	10.8	16.7	18.3	87.5	76.7
jennifer_philmon	34.4	29.2	37.5	97.9	63.5	theodora_tong	27.6	36.2	38.2	88.2	60.5
john_brown	22.9	30.2	29.2	94.8	71.9	vanessa_davis	19.2	20.0	20.8	90.0	70.8

Table 16: Per-persona TSAcc summary (%, persona-macro across the eight evaluated models). Gap is `oracle` minus `default`.

H Worked Trajectory Examples

This section gives concrete, trace-level examples rather than an aggregate failure analysis. Each example is drawn from a full human-readable evaluation trace; the left panel compresses the interaction, and the right panel records the Router/Classifier or evaluator state.

H.1 Learning-Session Examples

LS Example 1: Question Hits a Retained Rule

Compressed interaction trace	Scaffold reading
Source. One `atr`/Gemini 3 Flash Preview learning session. The user asks for help replying to a message thread. The agent calls `search_messages`, finds one thread, and asks the user what the reply should say. The user provides reply points. The agent calls `draft_message`, asks for approval, receives approval, and then calls `send_draft`. After the task is complete, it asks whether messages in this category should always be drafted first for review.	**Routed question.** The agent asks whether future messages of the same category should be drafted for review before sending. **Router.** RULE, strict=true. **Classifier.** HIT: `communication_01`. **Canonical user answer.** The user always wants editorial or culturally sensitive messages drafted first so they can revise them before they go out. **Close.** The simulator returns that answer, the agent acknowledges the standing rule, and the LS ends.

Table 17: A learning session where the post-task standing-rule question maps to a retained rule.

H.2 Test-Session Examples

TS Example: Wrong Argument on the First Relevant Action

Compressed interaction trace	Evaluator reading
Source. One `atr`/Gemini 3 Flash Preview test session. The user asks the agent to handle an invitation after checking the relevant calendar for conflicts. The agent calls `list_events`, finds an overlapping event, later checks unrelated windows, then makes the invitation-response call with `response=accept` and later issues a rescheduling call.	**Required action.** The invitation-response tool. **Observed first relevant action.** The required tool is present, but the argument is wrong. **Gold argument.** `response=tentative`. **Actual argument.** `response=accept`. **Verdict.** Gold fail. This illustrates the TS matching rule used throughout ATRBench: a later repair does not matter if the first action touching the rule's action surface carries the wrong decision.

Table 18: A test session that fails even though the required tool appears, because the first relevant call carries the wrong enum argument.

I Prompts

We reproduce the evaluation prompts below with placeholders for dynamic context.

I.1 Data Construction Prompts

The seven prompts below drive the construction pipeline (Appendix A): standing-rule generation and two-axis QC, rule-bound test-session generation, QC trace simulation, and repair, and learning-session skeleton and fill. Double-brace placeholders (e.g. {{RULE_JSON}}) are filled at call time; few-shot pools and persona or tool context are elided where marked. Non-ASCII glyphs in the deployed prompts are transcribed to ASCII here for typesetting.

Rule Generation (rules/gen)

```
# Rule Generation

From the persona below, produce {{N_RULES}} standing
rules. Each rule is one sentence describing a long-term
preference of this user and must map cleanly to one tool
call from the tool list.

## What makes a good rule

1. **Grounded in the persona.** Point to a specific
phrase, behavior, activity, occupation, or community.
Generic adjectives ("creative", "diligent", "careful") do
 not count -- they fit anyone.

2. **Surprising default.** A good rule flips what a
generic helpful agent would otherwise do; if the agent
would take the same action anyway, the rule is dead
weight.

   Today's LLM agents have a strong cautious / non-
   destructive bias, so cautious-side wedges (archive-
   over-delete, pause-over-cancel, confirm-before-
   destructive-write, prioritize-important, track-after-
   ordering) are NOT surprising -- they are the default.

   Example: "Delete promo emails outright." (Agents
   default to archive; this rule pushes the destructive
   action.)

3. **Maps to one tool call.** See `check_type` below.

Skip candidates that do not cleanly map (feelings,
internal moods, things the tool list cannot express).

## Output

JSON list. Each entry:

```json
{
  "rule_text": "<third-person sentence; do not name tools
   or parameters>",
  "canonical_answer": "<first-person sentence that stands
   on its own (e.g., 'I always X for Y' / 'When Y, I do Z
  ' / 'For X, I prefer Y'); do not name tools or
  parameters>",
  "counterfactual_default": "<what someone without this
  rule would naturally do>",
  "evidence": [
    {"type": "direct", "content": "<exact phrase from the
     persona>"}
    // OR {"type": "inferred", "content": "<reasoning
    chain rooted in specific persona facts>"}
  ],
  "check_type": "tool_identity | param_id | param_enum |
  confirm",
  "action_step": {
    "tool": "<exact tool name from tool list>",
    "param": "<param name on that tool, OR null, OR --
    for confirm rules -- the inner mutate tool name>"
  }
}
```

`check_type` -- pick one of four

- **`tool_identity`** -- pins WHICH TOOL. `param = null`.
- **`param_enum`** -- pins an ENUM VALUE on an enum-typed
 param.
- **`param_id`** -- pins WHICH ITEM (by attribute) on a
`*_id` / `*_ids` parameter. Not for ids the user hands in
 ("delete file X" -- that is instruction following).
- **`confirm`** -- asks for `get_user_confirmation`
before a mutate; `param` carries the INNER mutate tool
name (NOT a parameter on `get_user_confirmation`). A
confirm rule may also pin an inner-param preference; that
 surfaces later as test-session `gold_value`, not in `
action_step`.

Hard rules

- Tool and param names are exact strings from the tool
list.
- `param` is what the rule pins (output choice), not what
 the user hands in (input id). For instance, pin `
selected_alternative_stop_ids`, not `disrupted_item_id`.
- Do not put a *rejected* tool in `action_step`. For "
draft instead of send", `tool=draft_message`, never `
send_message` -- the rejected tool belongs in `
counterfactual_default`.
- For "do X, don't do Y" rules, X and Y must be
alternatives -- calling X precludes Y, or Y is something
nobody would call here.

  Example: "archive instead of delete" -- archive and
  delete are exclusive cleanup actions.

  Counter-example: "label community emails, don't archive
  " -- labeling and archiving are independent steps, so
  the agent does both anyway.

- No duplicate or contradicting bindings. "Direction" =
which side the rule prefers. At most one rule per `(tool,
 param, direction)`. Two rules on the same `(tool, param)
` are OK only if their directions are orthogonal axes (
one pins cuisine, another pins seating -- both can hold
on one restaurant call). Two rules pinning the same
trigger to opposite values: drop the weaker one.

Tool pairs that look similar -- pick deliberately

- `plan_trip` (new) vs `replan_trip` (existing trip;
needs `trip_id` + `disrupted_item_id`).
- `modify_event` (anything except time) vs `
reschedule_event` (time only).
- `update_tracker` (append to existing; needs `tracker_id
`) vs `create_document` (new doc / spreadsheet / note).

Examples

```json
// tool_identity
{"rule_text": "When cleaning up work files, archive
rather than delete.",
 "check_type": "tool_identity",
 "action_step": {"tool": "archive_files", "param": null}}

// param_enum
{"rule_text": "When an invite conflicts with something
unresolved, respond tentatively rather than declining.",
 "check_type": "param_enum",
 "action_step": {"tool": "respond_to_event_invite", "
 param": "response"}}

// param_id
{"rule_text": "For garden purchases, favor locally-
adapted native plants over generic landscaping options.",
 "check_type": "param_id",
 "action_step": {"tool": "place_order", "param": "
 product_id"}}

// confirm
{"rule_text": "When new fad-style health services come up
, check with her before booking.",
 "check_type": "confirm",
 "action_step": {"tool": "get_user_confirmation", "param
 ": "book_service_appointment"}}
```

Tool list
```

```
{{TOOLS_TEXT}}

## Persona

{{STRUCTURED_PERSONA}}

## Final per-rule checks

1. `counterfactual_default` names a *different action*,
not just less of the same.
2. The same persona trait is not already covered by
another rule in your output.

Output JSON list only. No code fence. No other text.
```

Rule Quality Control (rules/qc)

```
# Rule QC

Score each rule on two independent axes (`counter_default
` and `binding_sound`). The downstream keep / remove tag
AND-s them -- you only emit the per-axis labels and
reasons. Score each rule independently; you do not need
to consolidate duplicates, contradictions, or evidence
quality across rules (all handled elsewhere). Score using
 `rule_text`, `counterfactual_default`, `check_type`,
and `action_step`.

**Terminology.** *passive* = a generic helpful agent that
 does NOT know this rule. Note that `rule_text` is
pipeline metadata; passive does NOT see it -- at test
time passive sees a neutral instruction generated from
the rule + tool list + ontology defaults, so judge
passive from the underlying scenario rather than from `
rule_text` wording. *gold action* = the tool call this
rule wants the agent to take (i.e. `action_step`).

---

## `counter_default` (yes | no)

> If we strip this rule away, would a generic helpful
agent in the same scenario take the gold action anyway?

Yes -> `counter_default = no` (rule is redundant, no
personalized wedge). No -> `counter_default = yes` (real
wedge).

**NOT real wedges** (mark `no`). Current LLMs have a
strong cautious / non-destructive default, so cautious-
side rules are what passive does anyway: any "archive /
pause / preserve / keep" rule that picks the safer option
, and any "confirm before a destructive write" rule.

**OK wedges** (do NOT mark `no`). Rules that flip the
passive default or pin a value passive would not reach
for.

Example: "draft instead of send directly" -- passive
defaults to sending the prepared message; the rule flips
that.

Example: "respond=tentative on conflict" -- passive picks
 accept or decline; the rule pins the third enum value.

**Subtle failure.** Rule says "do X, don't do Y" but X
and Y are not mutually exclusive -- passive does both, so
 gold X is called even without the rule -> `
counter_default = no`.

Counter-example: "label community emails, don't archive"
-- labeling and archiving are independent steps; agents
do both anyway.

### Examples for this axis

{{COUNTER_DEFAULT_POOL}}

---

## `binding_sound` (yes | no)

> Knowing this rule, would an agent naturally land on
this exact `(check_type, action_step)`?

Yes -> `binding_sound = yes`. No -> `binding_sound = no`.

**Looks right but fails** (mark `no`):

- Rule is a cross-tool choice ("reminder rather than a
new calendar event") but the binding lands on a param
INSIDE one tool (e.g. `set_reminder.cadence`) -- an agent
 following the rule would care about which tool, not
which cadence.
- Rule talks about a dimension (time, formality, theme)
but the binding is `tool_identity` with `param = null` --
 no parameter slot carries the dimension, so calling the
tool does not express the preference.

**Looks wrong but actually OK** (do NOT mark `no`):

- Cross-tool wedge bound to the gold side -- rule "prefer
 A over B", binding `A, tool_identity`. The rule's action
 IS calling A; the rejected B belongs in `
counterfactual_default`, not in the binding.
- `param_id` rule whose dimension lives in candidate `
attributes` -- rule talks about time / theme / type,
binding is `*_id`. The runtime populates each candidate's
 `attributes` with that dimension and the agent selects
by reading them, so the binding carries the dimension
indirectly.

**Enum catch-all warning.** Avoid `param_enum` whose enum
 includes an `"all"`-style catch-all. Agent safety bias
favors the catch-all over the rule's specific value,
making the rule unreliably testable.

### Examples for this axis

{{BINDING_SOUND_POOL}}

---

## Rules to review

```json
{{RULES_JSON}}
```

Output format

```json
[
  {
    "rule_text": "<copy verbatim from input>",
    "counter_default": "yes" | "no",
    "counter_default_reason": "<one line; if 'no', name
    what passive would do by default; if 'yes', write '
    passes'>",
    "binding_sound": "yes" | "no",
    "binding_sound_reason": "<one line; if 'no', name
    what the rule asks for vs what the binding lands on;
    if 'yes', write 'passes'>"
  }
]
```

Output count = input count. Every input rule appears
exactly once.

Output JSON list only. No code fence. No other text.
```

Test-Session Generation (test_sessions/gen)

```
# Test Session Generation

Generate one test session for the given rule. At runtime
the session is user-offline -- the agent executes alone,
with no one to clarify with -- so the instruction must be
 self-contained and must NOT reveal the rule direction.

The session is valid iff the binary discrimination holds:

- **oracle** (knows the rule) hits gold tool + gold args.
```

- **passive** (does not know the rule), reading the same instruction + refs and reasoning by commonsense, does NOT hit gold.

Use `rule.counterfactual_default` as your prediction of passive's behavior and point your decoy design that way so the oracle / passive gap stays sharp.

This is a single-call task -- no refine loop. Reason through the steps below internally, then emit the final JSON.

Locate the discriminative dimension

From `rule_text` + `canonical_answer` + `counterfactual_default`: what does oracle do (gold)? what does passive do (counterfactual)? where is the difference -- `param_id` (which ref selected) / `param_enum` (which enum value) / `tool_identity` (which tool)?

Refs design

Per-decoy pull (param_id only)

Plan 1 gold + >=2 decoys. Every decoy needs at least one concrete *pull attribute* -- a property that, on its own, attracts a rule-blind agent. Menu: lowest price / highest rating / most popular / official or authoritative / most recently updated / closest distance / most full-featured.

- Decoy check: "Without the rule, which attribute pulls toward this decoy?" If you cannot name it, redesign -- a parallel choice with no pull leaks because passive picks at random.
- Gold check: "With the rule, does the rule-aware agent uniquely lock on gold?" If a decoy also satisfies the rule + instruction demands, oracle could legitimately pick it -- redesign.

For `param_enum` / `tool_identity`, refs still need >=1 gold + 1-2 decoys for context, but gold determination depends on the enum value / tool, not on the ref set.

Refs-as-a-whole must lean toward the counterfactual

Read `counterfactual_default`, work backwards to what signal in refs points that way, then amplify it. The rule-direction signal should be the *least* prominent.

- **tool_identity** (rule = `create_document`, counterfactual = `update_tracker`): refs must include 1-2 appendable-looking trackers so passive leans toward update.
- **param_enum** (rule = `scheme=by_project`): each ref carries the rule dimension AND other dimensions (`created_date`, `doc_type`) so passive has multiple classification paths.
- **param_id**: the rule-direction attribute must NOT be gold-exclusive. Give >=2 candidates with the rule's direction (1 gold + >=1 decoy), and let decoys beat gold on a secondary dimension (rating, distance, popularity) so passive picking by secondary signal lands on a decoy.

  Counter-example. Rule "same theme (nature)" + refs = 1 nature garden + 3 art / spa / cafe. Passive sorting "same theme" has no choice but gold.

  Example. Gold = garden (nature); decoy1 = overlook (nature, far); decoy2 = temple (cultural, 4.8 rating); decoy3 = cafe (4.9). Two nature candidates with secondary-dimension reversal.

Self-check: count refs carrying each rule-direction attribute. Only 1 (gold-exclusive) -> add a same-direction decoy.

Two more ref traps

- **Hard-constraint exclusivity** (`param_id`). If instruction lists hard constraints ("Old Harbour + 2 adults + private bathroom"), every decoy must satisfy all of them, else instruction-filtering rejects them and gold becomes the only valid pick. Differentiate decoys on secondary dimensions, not on the hard constraints.
- **Project-lifecycle naming** (`param_enum scheme=by_project`). Ref names that read like one project's stages (survey -> costs -> recap -> packet) commonsense-pin passive on `by_project` even with neutral instruction. Mix names across multiple visible projects, vary `file_type`, and spread `created_date` over months.

Gold triple-filter

All three must be yes before fixing gold:

1. **Rule-conformant**: gold satisfies every dimension of the rule direction. (Do not misread "A over B" -- gold must BE A, not just "not B".)
2. **Instruction-conformant**: gold satisfies the instruction's explicit non-rule demands (time, location, count, budget).
3. **Decoy disadvantage**: every decoy fails at least one rule or instruction dimension strictly worse than gold; no decoy simultaneously satisfies 1 + 2.

Instruction

Write a natural one-paragraph user request. Include the non-rule task parameters; do NOT include rule-direction words, the gold ref id, or the gold tool name.

Wording neutrality (the main passive-leak source)

For every concrete noun and verb, ask: "On its own, does this word push passive toward gold?" If yes, swap for neutral phrasing. Common leak patterns and the neutral rewrites:

- **Type-word** ("project files" when rule is `by_project`) -- neutralize to "random files that have piled up".
- **Action-word** ("draft a message" when rule is draft-over-send) -- neutralize to "send X a quick note".
- **Container / location-word** ("don't leave them in the workspace" when rule is archive-over-delete) -- neutralize to "clean them up".
- **Frequency / cadence-word** ("every week I do this" when rule is weekly cadence) -- neutralize to one absolute time ("Wednesday at 8pm"), and let oracle infer weekly from the rule.
- **Business-type word** ("claims, renewals, policy-change files" when rule is `by_project`) -- neutralize to "files I've collected over the past few months".
- **Replacement-need word** ("the compact one won't fit" when rule is exchange-over-refund and refs hold obvious replacements) -- neutralize to "I want to send this back".
- **Loaded-scenario** (long-haul "Seattle -> London, pick a seat" pre-commits passive to `window`; broken-item scenarios pre-commit `resolution_type='exchange'`; deadline-loaded messages pre-commit `priority='high'`; one-off scheduled tasks pre-commit `cadence='before_event'`) -- replace with a neutral scenario, or omit the cue field entirely.

Rule of thumb: as passive, read instruction alone -- does any word make you think "ah, must be option X"? If X = gold, drop the word. Legitimate task parameters (time, location, headcount, target person) must still be written clearly -- these are task context, not direction words. Test: remove the word; does the task still make sense? Yes -> keep; No -> it was a direction word, drop.

Discoverability

For every ref id in `gold_value` (or in `*_id` / `*_ids` slots inside `gold_value` for confirm rules), the instruction must contain at least one token that loose-matches a token in one of the ref's searchable attributes (`name`, `title`, `subject`, `sender`, `location`, `region`, `city`, `address`, `destination`, `origin`, `cuisine`, `service_type`, `category`, `doc_type`, `tags`). Without this, the agent's search returns 0 results and the session is unsolvable.

Loose-match = after lowercasing and splitting on whitespace / punctuation, some instruction token equals or is a substring of some attribute token. If persona / refs use English place names while the instruction is

Chinese (or vice versa), plant the English token verbatim somewhere in the instruction.

Time handling

ATRBench does not model a time axis: search tools do NOT take time / date filters. The agent fetches refs first, then reads `date_time` / `start_time` / `departure_date` on each candidate.

- Default: no specific time in instruction ("book dinner for me").
- Relative time is forbidden: never write "next Thursday / this week" or any non-Latin-script equivalent -- runtime hides the current date.
- Absolute date is allowed only for time-related rules (cadence enum, time-triggered notification, scheduling-conflict pattern): use `YYYY-MM-DD`, and the gold ref's date field must contain this date as a substring (`"2026-05-18T15:00:00"` matches `2026-05-18`).

When in doubt, omit time.

Derived entities

When the gold tool operates on a derived entity (orders / trips / reservations / bookings / appointments), the env hydrates the entity from the corresponding primary ref's attributes (see `derived:` sections in REF_SCHEMA for which primary ref carries which ID attribute). Place the derived ID on the right primary ref's `attributes`, else gold fails with "not found".

In the instruction, describe the entity by role / context, never by its specific name, and never with a literal id (`ord_xxx` / `rst_xxx`).

Example. "Return the candle I ordered last month." (Role.)

Counter-example. "Return the Cedar Glow Candle I ordered." (Specific name -- passive can match it directly.)

Few-shot examples (same / similar bucket)

{{FEW_SHOT_EXAMPLES}}

Counter-example -- real failure, do not repeat

`param_id` / numeric pull too obvious. Rule "prefer high-rating reliability over distance". Wrong refs: gold = Cascade Code (rating = 4.9, dist = 6.4 km); decoys top out at 4.7 rating with distances under 1 km. Passive locks onto "highest rating" = gold because LLMs are far more sensitive to rating jumps (4.7 -> 4.9) than to distance gaps (0.5 -> 6.4 km). Fix: mix gold's rating into the middle pack, and let a decoy own "highest rating + closest" so passive is pulled away.

Output JSON schema

You write 3 fields (plus optional `list_match_mode`):

```json
{
  "instruction": "<one English paragraph; non-rule task params present, no rule-direction words>",
  "gold_value": <see dispatch table below>,
  "list_match_mode": "strict" | "subset", // OPTIONAL -- only when gold_value is a list
  "references": [{"id": "...", "type": "...", "attributes": {...}}]
}
```

You do NOT write `domain`, `local_env.tools`, `session_id` / `session_type` / `persona_id` / `rule_id` / `rule_ref`, or `labels` -- the caller derives them.

`gold_value` dispatch (by `rule.check_type` x param type)

Find `<rule.param>`'s type in the tool signature inside `DOMAIN_TOOLS`.

`check_type`	`<rule.param>` type	`gold_value`
`tool_identity`	(n/a, no param)	`null`
`param_id`	`string` (single id)	`"<ref_id>"`
`param_id`	`array[string]` (list of ids)	`["<ref_id>", ...]` (typically length 1)
`param_enum`	`enum[...]`	`"<enum_value>"`
`confirm`	(param = inner mutate tool name)	`null` (confirm-only) OR `{<inner_param>: <inner_value>}`

`list_match_mode` (emit only for `array[string]` `param_id` rules):

- `"strict"` (default): `set(gold) == set(actual)`, no extras. Use when rule intent is "only these qualify".
- `"subset"`: `set(gold) subseteq set(actual)`, gold ids must all appear and extras are tolerated. Use when rule intent is "always include these".

When in doubt, default to `"strict"` (you may omit the field).

Confirm rule details

`rule.action_step.tool` is always `get_user_confirmation`; `rule.action_step.param` carries the inner mutate tool name (e.g. `"delete_files"`, `"book_service_appointment"`).

- No inner-param preference (purely "confirm vs not"): `gold_value = null`. Caller fills `arguments = {target_tool: <inner>}` and `compare_args = ["target_tool"]`.
- Inner-param preference (e.g. "ask before canceling the recently-added appointment"): `gold_value = {<inner_param>: <value>}`. Caller adds `target_params: <gold_value>` and `compare_args = ["target_tool", "target_params"]`. The inner param must exist on the inner mutate tool's signature with a safe type (`*_id` / `*_ids` / `enum` / `boolean`).

Hard self-check (run before emitting)

1. Non-rule task params (location, headcount, target person) are written clearly in the instruction.
2. References contain exactly 1 object satisfying the rule direction for `param_id` (and inner-`param_id` for confirm rules).
3. `gold_value` matches `rule.check_type` x `<rule.param>` type per the dispatch table.
4. Ref attributes hold only neutral facts (numbers, categories, identity, time, participants) -- no value-judgments, recommendations, or actionability hints.

Confirm rule special constraint

If `rule.check_type == "confirm"`, the instruction is an ordinary user request ("help me schedule X" / "handle it") and must NOT contain any wording about a confirm flow ("ask me first", "check with me", "confirm before", "get my permission", "decide later") or any two-step process narration ("first X, then Y"). The confirm step is what oracle infers from the rule -- if instruction leaks it, passive also calls `get_user_confirmation` and the discrimination disappears.

id-reference hard red lines (static check enforces)

1. Any `*_id` / `*_ids` value in `gold_value` must be the id of an object in `references`. No fabricated ids.
2. `references[i].type` must be a primary type listed in REF_SCHEMA (the `type:` headers -- not the `derived:` sections).

Inputs

Rule

```json
{{RULE_JSON}}
```

Domain = `{{DOMAIN}}` available tools

```json
{{DOMAIN_TOOLS}}
```

Domain `{{DOMAIN}}` reference schema

{{REF_SCHEMA}}

Emit strict JSON now, no markdown code fence.

Test-Session QC: Agent Trace Simulation (test_sessions/qc)

Test Session QC -- Agent Trace Simulation

You are an assistant. Complete the user's task by calling the available tools. Output the full trace in a single response -- list every tool call you intend to make, in order. Do not split across turns; do not wait for tool returns.

{{RULE_CONTEXT_BLOCK}}

Task

User request

{{INSTRUCTION}}

Available tools

```json
{{TOOL_SPECS}}
```

Objects in the environment

```
{{REFERENCES}}
```

> Candidate objects you may reference by id. Pick directly -- do NOT add a `search_*` / `list_*` discovery prologue.

Conventions

- List tool calls in natural execution order.
- Argument values must conform to the schema: `enum` values come from the listed set (never invent); `boolean` is `true` / `false`; other `string` params must not be fabricated; `id` params may reference only the ids listed above.
- Always supply required parameters; for optional parameters, supply a value only when the user request, the environment objects, or an applicable user preference clearly determines it.

Output format (strict JSON, no code fence)

```jsonc
{
  "trace": [{"tool": "<tool_name>", "arguments": {<kwargs>}}, ...],
  "reason": "one sentence explanation"
}
```

Output now.

Test-Session Repair (test_sessions/refine)

Test Session Repair

The previous test session failed QC. Rewrite a new version (strict JSON, no code fence) such that, simultaneously:

- **Binary discrimination holds**: oracle hits gold; passive (commonsense) does NOT.
- The agent's first tool call hits the rule's relevant tool (the rule's decision point is not at a side step).
- Instruction is self-contained: every non-rule task parameter is written clearly.
- Refs hold only neutral facts (numbers / categories / source / identity / time / participants) -- no value-judgments, recommendations, or actionability hints.
- For `confirm` rules, the instruction is an ordinary user request and must not leak the confirm flow (no "ask me first", "check with me", "confirm before", or two-step process narration).

Repair is NOT patching -- you may change any field.

Cross-reference `failure_reasons` with the N `upper_reasons` / `lower_reasons` and their per-sample traces and match flags. Keywords repeated in passive self-reports point to where instruction or refs leak direction.

Static-error precise fixes

- `instruction_leaks_ref_id:<ID>` -> replace the literal id string with a natural-language description ("the train ticket from Florence to Venice" instead of `gt_florence_venice_20260512`).
- `action_{i}_id_missing_from_refs:<tool>.<arg>=<id>` -> `gold_value` references an id not in `references`. Either add the object (`{id, type, attributes}`) or change `gold_value` to an existing id.
- `ref_type_invalid:<type>:domain=<domain>` -> swap to a primary type from REF_SCHEMA's `#### type:` headers (NOT the `derived:` sections, which are auto-hydrated).
- `action_step_tool_is_exploration_prefix` -> rule-level error; the repair stage cannot fix it -- return to rule generation.

Reading the QC failure mode -> targeted fix

- **Mode A** (`upper_matched = False`): oracle did not hit gold. Causes: refs lack an object satisfying the rule direction (fix `references` so exactly 1 ref satisfies every dimension); `gold_value` is not what oracle would pick (fix `gold_value`); or the instruction's task parameters conflict with the gold ref's attributes ("Beijing restaurant" but gold ref is Shanghai -- align one with the other). Read `upper_reasons` for the sentence where oracle says why it picked X instead of gold.

- **Mode B** (`upper_matched = True, lower_matched = True`): the main failure mode -- passive also hit gold. Find repeated keywords in `lower_reasons`:

  - "I picked X because the instruction said Y" -> instruction leak. Apply the wording-neutrality guidance from the generation prompt (drop the type / action / container / cadence / business-type / replacement-need / loaded-scenario leak).
  - "I picked X because its attribute Z fits best" -> refs let gold's rule-direction attribute be too prominent. Apply the refs-as-a-whole guidance from the generation prompt: give decoys pull attributes (price / rating / popularity / official / newest / closest / most full-featured); for `param_id`, if gold is exclusive on the rule attribute, the decoys must beat gold on a secondary dimension; for `tool_identity`, refs must include candidates the counterfactual tool would handle.

- **Mode C** (`upper_matched = False, lower_matched = True`): both wrong. Usually gold or markers were set wrong -- fix oracle per Mode A; if the lower problem does not disappear as a side effect, apply Mode B.

Real-failure reminders -- do not repeat

- `param_id` / gold-exclusive on the rule direction -> add 1-2 same-direction decoys that beat gold on a secondary dimension.
- `param_id` / numeric gap too obvious (gold rating 4.9 vs. decoys topping out at 4.7): LLMs lock onto "highest rating" = gold; gold rating must NOT be the maximum.
- `param_enum` / task-word leak ("project" ~= `by_project`; "key" ~= critical): neutralize phrasing.
- Rule direction == LLM default: fundamentally unconstructible -- do not force-fix, report "unconstructible".

Few-shot examples (same / similar bucket)

{{FEW_SHOT_EXAMPLES}}

Inputs

Previous test session

```json
{{CURRENT_TASK}}
```

Failure reasons

```json
{{FAILURE_REASONS}}
```

Rule

```json
{{RULE_JSON}}
```

Domain = `{{DOMAIN}}` available tools

```json
{{DOMAIN_TOOL_SPECS}}
```

References format contract

{{REF_SCHEMA}}

Hard MUSTs (violation -> entire repair is discarded)

Retain these 3 fields; values may change but keys cannot be dropped:

1. `instruction` (str, non-empty).
2. `references` (>=3 objects for `param_id`; smaller is OK for other check_types if there is at least 1 gold-context object).
3. `gold_value` (per `rule.check_type`):
   - `tool_identity` (mutate, `action_step.param = null`) -> `null`
   - `param_id` -> `"<gold ref id>"` (must exist in `references`)
   - `param_enum` -> `"<enum value>"` from the enum's value set
   - `confirm` -> `null` (confirm-only) or `{"<inner_param>": <value>}` (inner_param must be safe-typed on `rule.action_step.param`'s tool signature)

Do NOT output `domain`, `local_env.tools`, `labels`, or `session_id` -- the caller derives them. A common past mistake was emptying or mistyping `gold_value`; do not repeat it.

Output the complete repaired test session JSON now, no markdown code fence.

Learning-Session Skeleton (learning_sessions/skeleton)

Learning-Session Skeleton Generation

You are a scenario architect for AI-agent evaluation data. Given a persona's structured profile, produce {{N}} learning-session skeletons. Each skeleton has two fields only:

- `domain` -- one of the 6 supported domains
- `theme_one_line` -- one English sentence describing what this persona, in everyday life, asks an AI agent to do

A downstream fill stage fills in the full session content (instruction, task_params, references, expected_tools). Your job is pool-level scenario architecture only -- you commit to *what* each session is about, not *how* the agent will execute it.

Generation principles

1. **A daily-interaction trail.** The {{N}} entries together form a stretch of this persona's interactions with the agent. Order them in natural temporal order (earliest first); the caller stamps `day_offset` from the index. Adjacent entries may carry causal or topic continuity (booked weekend gathering -> drafted thank-you note; planned business trip -> booked flight -> booked hotel) -- 2-4 such chains is enough. Do NOT batch entries by domain (first 5 reservation, last 5 workspace) -- real life interleaves.

2. **Domain distribution follows the persona's actual life.** You are not required to cover all 6 domains. Read the persona's occupation, lifestyle, and social_context: an art enthusiast may have lots of commerce; a commuter office worker may lean on communication / scheduling; a homemaker may use reservation more. Irrelevant domains can be entirely absent -- do not shoehorn.

3. **Bake situational variety into theme text.** Since you emit one sentence per session, vary time-of-day, day-of-week, location, and mood through wording. Mix early-morning errands, weekend evenings, rushed weekday afternoons, in-flight downtime, and quiet at-home moments.

   Example. "Triage my work inbox before today's 9am steering check-in."

4. **Theme-implied tool coverage.** Each theme implicitly invokes downstream tools. The {{N}} themes must collectively touch varied tools -- do not converge 15 themes on "search restaurant + book restaurant". Use the capability table below as a sanity-check; do not propose asks no domain supports ("redecorate my living room").

5. **Result-object reachability (commerce / travel / reservation).** A learning session is single-session: no prior turn the agent can rely on to know an existing `order_id` / `trip_id` / `booking_id` / `appointment_id` / `subscription_id`. The current tool set has no `search_orders` / `search_trips` / `search_bookings` / `search_appointments`, so for these three domains the agent cannot fetch an existing result object from a free-text description.

   Counter-example. "Track the bird-seed order I placed for the cardinals." (Commerce; no `search_orders`.)

   Example. "Order another bag of cardinal blend so the feeders do not run out." (Reframed to a NEW `place_order`.)

   This restriction applies ONLY to commerce / travel / reservation. workspace (`list_files`, `search_documents`, `list_trackers`), scheduling (`list_events`, `check_conflicts`), and communication (`search_messages`, `list_labels`) all expose search / list tools, so "update / modify / archive / triage" themes are fine there.

```
  > Note: this LS restriction does NOT apply to TS -- TS
   hydrates derived entities from primary ref attributes.
   LS is first-contact with no hydration.

---

## The 6 available domains

- `commerce` -- shopping / orders / order management /
subscriptions / returns
- `reservation` -- restaurant bookings / event tickets /
local-service appointments
- `travel` -- flights / hotels / trip planning / ground
transport
- `communication` -- email triage / priority / archive &
label / drafting replies
- `scheduling` -- calendar events / reminders / meetings
/ conflicts
- `workspace` -- file organization / documents /
archiving / project tracking

## Domain capability table (sanity-check; do NOT name
tools in theme text)

{{TOOLS_BRIEF}}

---

## Field spec

- **`domain`** -- pick 1 of 6.
- **`theme_one_line`** -- one English sentence. Carry
enough situational color (time, occasion, location hint,
mood) for the fill stage to ground the fill realistically
. Do not spell out specific param values that should be
elicited at runtime (concrete date / time, exact party
size, specific cuisine, exact ids) -- those are for the
fill stage. Do not name tool functions or domain words
verbatim.

  Example. "Book a quiet Saturday dinner spot to
  celebrate my sister's birthday."

  Counter-example. "Book a Japanese place in the East
  Village for next Tuesday at 7pm, 4 people." (Leaks
  cuisine, location, date, party size.)

---

## Input

### Persona structured data

{{STRUCTURED_PERSONA}}

---

## Output

Strict JSON array of {{N}} elements, no code fence and no
 markdown tags:

```json
[
  {"domain": "<one of 6>", "theme_one_line": "<one
  English sentence>"},
  ...
]
```

Do NOT output `session_id` / `day_offset` (the caller
stamps them) or `situations` / `references_sketch` / `
expected_tools` (those moved to the fill stage).

Self-check (before emitting)

- Exactly {{N}} entries, each with `domain` in {commerce,
 reservation, travel, communication, scheduling,
workspace}.
- `theme_one_line` is one English sentence, no tool names
, no over-specific param values.
- Sessions interleave across domains and span varied time
 / mood / location through wording (Principles 1 and 3).
- Themes fit the persona and land within some domain's
capability (Principles 2 and 4).
- No commerce / travel / reservation theme starts from an
 already-existing order / trip / booking / appointment /
subscription (Principle 5).
```

Learning-Session Fill (learning_sessions/fill)

```
# Learning-Session Fill

The skeleton stage produced a thin skeleton (`domain` + `
theme_one_line`). Fill in one complete learning session
from that skeleton.

## Core constraints

### Write daily interactions only -- do not reveal
preferences

You hold no long-term rule pool. Write only factual
scenarios (one specific thing this persona is handing off
 to the agent right now). Do not hint, in instruction /
task_params / references, at "this person generally
prefers X" or any cross-task decision direction.

### Lazy-user contract

- **`reason_for_call`** is what the user holds in mind as
 the overall reason for contacting the agent -- only `
user_sim` reads it; the agent never sees it. user_sim
uses it to generate the opening message naturally (vague,
 direction-only).

  `reason_for_call` MUST be vague: convey direction /
  occasion / mood, but must NOT contain any specific `
  task_params` value (cuisine, category, location, brand,
   person name, event title, project name, file / folder
  name, document type, or any concrete identifier).
  Generic container words any user might say upfront ("
  inbox", "calendar", "tracker", "folder", "trip") are
  fine; specific content words that name *this* user's
  target are not.

- **`task_params`** is the user's full mental list of
specific needs, a flat dict `{field_name: concrete_value
}`. Value types: numbers, dates, strings, ids, enum
literals, lists -- whatever the field needs. Do NOT add
desc / rationale fields; user_sim will paraphrase at
runtime.

- Runtime rule for user_sim: when the agent asks about a
`task_params` field, paraphrase the value; otherwise
reply "I don't know / whatever you think". Repeat the
answer if the same field is asked twice.

### Closed-loop traceability (HARD invariants -- every `
task_params` key needs a runtime destination)

Every key falls into exactly one of these destinations:

| Destination | Key shape | Value constraint |
|---|---|---|
| A. Direct tool argument | matches a parameter name on a
 tool the session expects to use | type matches; if enum,
 value in allowed set |
| B. Search filter against refs | matches a `local_env.
references[*].attributes` key on the matching ref type |
value MUST appear lexically in at least one reference's
attribute value |
| C. Reference id selector | ends with `_id` or `_ids` |
value(s) MUST exist in `local_env.references[*].id` |
| D. Drafting points | ends with `_points` / `_keywords`
/ `_bullets` | `list[str]`, each item <= ~80 chars; NOT a
 pre-authored body |

**Forbidden in `task_params`:**

- **E. Runtime autofill fields** -- `contact_info` / `
payment_method` / `shipping_address` / `traveler_info` /
 `guest_info`. These are auto-injected from
PersonaProfile; including them is redundant and will be
rejected.
- **F. Server-generated derived IDs** -- `order_id` / `
trip_id` / `booking_id` / `appointment_id` / `
reservation_id`. LS is first-contact; the user does NOT
```

know server IDs (`ord_xxx`). (TS hydrates derived entities from primary ref attributes; LS does not -- derived IDs in LS refs are also forbidden.) Use a `<noun>_description` field instead (e.g. `order_description: "the recent bird-seed order"`); the agent resolves the actual id via `track_*` / `search_*` at runtime.
- Algorithmic / procedural instructions encoded as values (`selection_rule: "choose the first conflict-free option ..."`). Values are *facts*, not *agent algorithms*.
- Output-control booleans (`include_daily_breakdown: true`). Agent output knobs, not user-side facts.

References pool must support `gold_trajectory`

The trajectory you write must be physically executable against your refs. If the trajectory contains a search step, refs must include >=1 of the corresponding type, or the agent stalls.

Granularity match

Any string-typed `task_params` value used as a search filter (location / cuisine / category / region) must appear lexically (token-level overlap) in at least one ref attribute. If the user's mental notion is at a different granularity than refs, use the granularity that lexically overlaps refs.

Counter-example. `task_params.location = "Upper Manhattan"` while refs say `location: "Washington Heights"` -- no token overlap, so the env's substring filter returns 0.

Example. `task_params.location = "Washington Heights"` matching `refs[*].attributes.location = "Washington Heights"`.

Inputs

Skeleton (the skeleton stage has set this; do NOT change domain or invent themes)

```json
{{SKELETON}}
```

Persona structured data (background, for grounding)

{{STRUCTURED_PERSONA}}

Domain `{{DOMAIN}}` available tool specs

```json
{{DOMAIN_TOOLS}}
```

Domain `{{DOMAIN}}` reference object schema

{{REF_SCHEMA}}

Output JSON schema

```json
{
  "reason_for_call": "<<=10 word verb+noun phrase>",
  "task_params": {"<field_1>": <concrete value>, ...},
  "local_env": {
    "references": [
      {"id": "<id with short suffix>", "type": "<valid ref_type>", "attributes": {...}},
      ...
    ]
  },
  "gold_trajectory": [
    {"tool": "<tool name from this domain>", "arguments": {...}},
    ...
  ]
}
```

> `local_env.tools` and `expected_tools`: you do not output these -- the caller fills `tools` with the whole-domain list and derives `expected_tools` from your `gold_trajectory`.

Field requirements

`reason_for_call`

Ultra-abstract phrase, <= 10 English words, expressing only "verb + general object". `user_sim` reads it and paraphrases it as the opening message; everything specific stays in `task_params` and is disclosed only when the agent asks.

Strictly forbidden in `reason_for_call`:

- Any `task_params` value, verbatim or as a synonym (cuisine / category / location / brand / event title).
- Any time-of-day / day-of-week / specific occasion / mood adjective ("this weekend", "before today's meeting", "to celebrate").
- Any "head start" that lets the agent skip asking.

Example. "Help me with a dinner reservation."

Counter-example. "Help me book a sushi place in SoHo for two later this week." (Leaks cuisine, location, party size, and date hint.)

Length check: if the draft has more than 10 words, strip everything except verb + general noun phrase.

`task_params` (3-10 fields, flat key -> value)

- Include ALL the information the agent needs to complete the task.
- Values are concrete (numbers, dates, place names, headcounts, ids, enum values) -- no placeholders.
- Every field must be answerable by user_sim when asked. Since the instruction reveals nothing, every field is obtained by the agent asking or searching.
- Every (key, value) satisfies one of destinations A--D in the traceability table above.
- Do not stuff agent output text into a value: no full body for `reply_body` / `document_content` -- use a `*_points` drafting list.
- **Executability**: think clearly about the last action tool the agent calls (`book_` / `place_` / `send_` / `create_` / `cancel_` / ...). Its required parameters must come from a value in `task_params`, an id from `local_env.references`, or a runtime auto-injected identity field (`contact_info` / `traveler_info` / `guest_info` / `shipping_address` / `payment_method` -- these MUST NOT appear in task_params).

`gold_trajectory` (oracle solution path -- 2-5 steps)

The sequence of tool calls an agent would make if it knew all `task_params` upfront (no asking). Each step is `{"tool": <name>, "arguments": <dict>}`. The caller derives unique tool names from this trajectory.

Hard requirements:

- Each step's `tool` is in this domain's tool list (`{{DOMAIN_TOOLS}}`).
- **Every concrete argument value** (required and optional, search filters and write payloads) must come from one of:
  1. A value in `task_params` (verbatim).
  2. An id from `local_env.references[*].id`.
  3. A placeholder `<authored from <field>[, <field>...]>` for long-text drafting args (every referenced `<field>` must exist in `task_params`).
  4. A runtime auto-injected identity field, OMITTED from arguments (env auto-fills).
  5. A recoverable broad-list default for narrowing filters the agent could safely omit: `search_messages.folder in {inbox, sent, drafts}` (omitting returns all non-archived messages); `list_events.calendar` (omitting returns all calendars). NOT recoverable: `search_messages.folder = "archive"` (archived messages are hidden by default -- this must come from task_params).
  6. An agent-side control flag (`sort_by`, `limit`, `language` defaults, output-shaping booleans, `field` / `update_action` selectors). These are how the agent

```
  invokes the tool, not user-side facts, and need not
  appear in task_params.
- Anything else (a `doc_type`, `cuisine`, `location`, `
sender`, `priority`, `title`, `start_time`, `
participants` invented only in `gold_trajectory`) is
oracle-only knowledge user_sim cannot disclose -- fail.
- Every `search_*` / `list_*` step must be answerable by
your refs (>=1 ref of the matching type satisfying the
step's filters).
- Every id used as an argument must exist in `local_env.
references` with the right type.
- The last step is typically the action tool. Trajectory
length 2-5 (search -> optional compare -> action).

For long-text drafting args (e.g. `body` on `
draft_message`, `content` on `create_document`), use a
placeholder like `"<authored from reply_points>"` -- do
not compose the full text.

**Examples** (replace `<task_params.X>` with the literal
values you wrote):

```json
// reservation
[
  {"tool": "search_restaurants", "arguments": {"cuisine":
   "<task_params.cuisine>", "location": "<task_params.
  location>", "sort_by": "rating"}},
  {"tool": "book_restaurant", "arguments": {"
  restaurant_id": "<gold ref id>", "date_time": "<
  task_params.date_time>", "party_size": 2}}
]

// communication
[
  {"tool": "search_messages", "arguments": {"folder": "
  inbox", "sender": "<task_params.target_sender>"}},
  {"tool": "set_message_priority", "arguments": {"
  message_ids": ["<msg id>"], "priority": "high"}},
  {"tool": "draft_message", "arguments": {"thread_id": "<
  thread id>", "language": "en", "body": "<authored from
  reply_points>"}}
]
```

`local_env.references` (3-6 entries)

- Each ref: `{id, type, attributes}`. `type` must be
valid in `{{REF_SCHEMA}}` -- do not invent. `attributes`
follow the schema contract. `id` follows the schema's
recommended prefix (`rst_`, `prod_`, `msg_`) with a
numeric suffix (e.g. `rst_momoya_01`).
- Cover every search-style tool in `gold_trajectory`: if
a step is `search_restaurants`, refs must include >=1 `
restaurant`; same for `search_destinations` -> `
destination`, `list_events` -> `calendar_event`.
- Diverse and natural distribution -- avoid 4 near-
identical items.
- References hold only pre-existing candidate objects
available before the agent starts. Do not put result
objects (orders already placed, bookings already made) in
 references; use a descriptive field like `
order_description: "the recent art-supply order"` and let
 the agent fetch via `track_*` / `search_*`.
- Derived IDs in ref attributes are LS-forbidden.
REF_SCHEMA's `derived:` sections describe TS-only env
hydration; LS is interactive first-contact with no
hydration, so do NOT put `order_id` / `trip_id` / `
booking_id` / `reservation_id` / `appointment_id` in any
 ref's attributes.

Domain-specific atomic constraints

scheduling -- `title` + `start_time` + `end_time`
must come from the SAME calendar_event ref

If `task_params` includes any of `title` / `start_time` /
 `end_time` for an existing event, they must jointly
match a single `calendar_event` ref's attributes. Treat
the triplet as one atomic write -- change one, sync the
other two. Picking the title from one ref and the time
from another (or made-up time) is the most common
scheduling-domain failure.

workspace -- reference doc_type uses a dedicated `
task_params` field

When `gold_trajectory.search_documents` uses a `doc_type`
 different from `task_params.doc_type` (the agent looks
at an existing template / reference before creating the
user's actual document), declare the search type under
one of: `source_doc_type` / `template_doc_type` / `
reference_doc_type`. If multiple reference types are
consulted, use a distinct field for each.

Example. `task_params: {doc_type: "note", source_doc_type
: "template", ...}` -> gold: `search_documents(doc_type="
template")` then `create_document(doc_type="note")`.

Counter-example. `task_params: {doc_type: "note", ...}`
-> gold: `search_documents(doc_type="template")` -- no `
task_params` field carries the reference type, so
user_sim cannot answer if asked.

JSON only -- no markdown code fence, no other text.
```

I.2 Agent Runtime Prompts

We show runtime prompt templates with placeholders for dynamic context. The domain-specific policy is listed separately in Appendix I.3; long <context> contents are elided.

Agent Learning-Session Prompt Template

```
<instructions>
You are the user's long-term personal assistant. You
serve this same user
across many tasks over time.

## Execution Conventions
- Any information must be obtained through search / list
/ track tools first
  -- never fabricate IDs.
- ID parameters must come from tool results, not from
names or keywords. See
  `<policy>` for the ID discovery path in each domain.
- Identity fields are auto-filled from the user's account
 defaults -- omit
  them even when the tool schema marks them optional, and
   never block on
  them: `shipping_address`, `payment_method`, `
  contact_info`,
  `traveler_info`, `guest_info`.
</instructions>

{DOMAIN_POLICY}

<context>
{omitted: cross-session context block, if non-empty}
</context>

<session_mode>
The user is online and will respond to your messages.

## Protocol
- Complete the user's request and tell the user when done
.
- When you lack key inputs to complete the current task,
ask the user rather
  than calling tools with guessed values.
- If you can't make progress -- empty search/list results
 after one or two
  reasonable attempts, or any other blocker -- ask the
  user rather than
  looping on tools.
</session_mode>

<send_protocol>
The user only hears you through the `send_to_user` tool.
Plain assistant text
outside this tool is internal reasoning only -- the user
will not see it.
</send_protocol>
```

```
{VARIANT_ADDITION}
```

Agent Test-Session Prompt Template

```
<instructions>
You are the user's long-term personal assistant. You
serve this same user
across many tasks over time.

## Execution Conventions
- Any information must be obtained through search / list
/ track tools first
  -- never fabricate IDs.
- ID parameters must come from tool results, not from
names or keywords. See
  `<policy>` for the ID discovery path in each domain.
- Identity fields are auto-filled from the user's account
 defaults -- omit
  them even when the tool schema marks them optional, and
   never block on
  them: `shipping_address`, `payment_method`, `
  contact_info`,
  `traveler_info`, `guest_info`.
</instructions>

{DOMAIN_POLICY}

<context>
{omitted: raw context block or oracle target-rule
statement, if non-empty}
</context>

<session_mode>
The user is offline.

Complete the user's request independently.

## Termination
- When the task is complete, or when no path forward
remains: must send a
  single `finish_session` tool call with no other content.

- On tool errors, try different parameters or a different
 tool before giving
  up.
</session_mode>
```

Variant-Specific Learning Additions

```
--- default ---
{empty}

--- atr ---
<standing_rule>
A standing rule:
- is a long-term user preference;
- pins how you should act on a recurring decision;
- holds across the user's future tasks;
- is not a specific instance from the task.
</standing_rule>

<standing_rule_acquisition>
You may ask about a standing rule of theirs that might
prove useful in
serving them later.

Asking costs the user's patience (cost); a question that
matches a real
rule of theirs reduces future errors (benefit).
</standing_rule_acquisition>

--- always_ask ---
<standing_rule>
A standing rule:
- is a long-term user preference;
- pins how you should act on a recurring decision;
- holds across the user's future tasks;
- is not a specific instance from the task.
</standing_rule>

<standing_rule_acquisition_required>
After you have largely finished the user's current task,
ask exactly one
question about a standing rule of theirs that may prove
useful in serving
them later. Do not ask more than one such question in
this conversation.
</standing_rule_acquisition_required>
```

Oracle Context Injection

```
The oracle variant skips learning sessions. In each test
session, the
same test-session prompt template is used, with the
context block containing
only the canonical answer for the target rule:

<context>
[Prior statement 1]
[user]: {canonical answer for the target rule}
</context>
```

I.3 Agent Domain Policy Prompts

Agent Domain Policy Prompts

```
--- domain_commerce ---
<policy domain="commerce">
## ID discovery paths
- `subscription_id` <- `review_recurring_charges` (no
search_subscriptions tool)
- `order_id` <- `search_products` result attributes,
previous `place_order`
  return, or instruction (no search_orders tool)

</policy>

--- domain_reservation ---
<policy domain="reservation">
## ID discovery paths
- `reservation_id` / `booking_id` / `appointment_id` <-
previous `book_*`
  return or instruction (no search_reservations /
  search_bookings /
  search_appointments tool)

## Shape constraints
- `track_reservation_updates`: exactly one of `
reservation_id` /
  `booking_id` / `appointment_id`

</policy>

--- domain_travel ---
<policy domain="travel">
## ID discovery paths
- `plan_trip.selected_stop_ids` /
  `replan_trip.selected_alternative_stop_ids` <- `
  search_trip_stops`
- `booking_id` <- previous `book_*` return
- `trip_id` <- previous `plan_trip` return

## Shape constraints
- `track_trip_updates`: exactly one of `trip_id` and `
booking_id`

</policy>

--- domain_communication ---
<policy domain="communication">
## ID discovery paths
- `label_ids` <- `list_labels`
- `message_ids` <- `search_messages`
- `thread_id` (for replying) <- `search_messages`
- `folder_id` (optional) <- `list_message_folders`
- `draft_id` <- previous `draft_message` return

## Shape constraints
- `draft_message` / `send_message`: at least one of `
recipient` and `thread_id`
```

```
</policy>

--- domain_scheduling ---
<policy domain="scheduling">
## ID discovery paths
- `event_id` <- `list_events`, `check_conflicts`, or
previous `create_event`
  return
- `target_id` (for `set_reminder` / `track_event_updates
` /
  `find_alternative_slots`) <- same sources as `event_id`

</policy>

--- domain_workspace ---
<policy domain="workspace">
## ID discovery paths
- `file_ids` <- `list_files`
- `folder_id` <- `list_file_folders`
- `tracker_id` <- `list_trackers`
- `document_id` <- `search_documents`
- `create_document`: no ID input needed

</policy>
```

I.4 User Simulator Prompts

User Simulator Opening Prompt

```
--- system ---
You play the user, having just contacted the task
assistant. Send the first message to open the
conversation.

- Write 1-2 conversational sentences.
- State just enough immediate task context for the
assistant to begin
  (for example, "haircut in Decatur" or "flight from
  Atlanta to Las
  Vegas").
- Leave follow-up details, choices, and execution
specifics for the
  assistant to ask about or look up.
- Do not reveal hidden/reference ids, exact selected
targets, exact
  titles/names, destination folder/list/project names,
  drafted content
  points, or long-term preferences.

Output only the opening line -- no explanation, quotes,
or prefix.

## Intent
{learning_session.reason_for_call}

## Opening hint (optional; sanitized from the first gold
step)
- `{arg_key}`: {arg_value}
...

--- user ---
Start the conversation.
```

User Simulator Reply Prompt

```
--- system ---
You play the user, talking with the task assistant.

You have one tool: `mark_task_complete` -- call it to
signal the session is done.

Your job is to answer task questions from your `##
Requirements` and help the assistant complete the current
 task.

## How to answer
- If the assistant asks about something covered in your
Requirements, answer from there -- match the way it asked
 (one detail if it asked for one; several together if it
asked for several).
- If the assistant asks about anything not in your
Requirements, you don't have an answer to give. Say
something like "no strong preference there -- your call"
/ "whatever works" / "up to you". Even when framed as a
binary choice ("A or B?"), do not pick A or B; picking
either side counts as making up an answer you don't have.
- If the assistant asks for help or seems stuck, use your
 `## Guided path` as a reference and tell it what to do
next in everyday language.

## Style
- Speak only when prompted; don't volunteer information
beyond what's directly relevant.
- Use everyday language; describe content, not tool names
, field names, or parameter names.

## Ending the session
Stay within your stated intent. The conversation shows `[
invoked tool_name(...) -> ok]` markers for actions the
assistant has completed; the final action in your `##
Guided path` showing such a marker is your signal that
the core task is done.

Keep replying normally while the assistant is collecting
details, asking questions, performing the task, reporting
 progress, or waiting for any user-visible answer from
you.

After the task is complete, close the conversation
naturally and gradually move it toward a normal ending.
Keep replying while the assistant is still responding to
that closing exchange.

Call `mark_task_complete` (in its own turn, with no text)
 only after the assistant has acknowledged the closing,
said goodbye, or made only a generic final offer of
further help.

If the task cannot be completed, terminate the same way
via `mark_task_complete`.

## User background
{persona runtime narrative}

## Intent
{learning_session.reason_for_call}

## Requirements
- `{task_param_key}`: {task_param_value}
  {optional referenced-object attributes}

## Guided path
1. {tool_name}({arguments})
...

--- messages ---
{prior conversation, with agent messages as user-role
messages and user-simulator replies as assistant-role
messages}

--- user ---
{agent_text}

--- optional user message on Router-positive turns ---
{rule_answer_hook}
```

I.5 Router and Classifier Prompts

The placeholder {{FEW_SHOT_EXAMPLES}} is replaced by the active shot bank before the scaffold LLM call.

Router Prompt

```
You decide whether one agent message asks the user about
a standing rule.

A standing rule is a user preference, habit, or default
that would shape
the agent's behavior on a different future task.

## Input
```

```
- `<reason>`: the agent's free-form intent for this turn.
 Treat it as
  the agent's own statement of why it is speaking.
- `<output>`: what the agent actually said to the user.

## Decision

Return `is_strict_rule_question = true` when `<output>`
contains a question
the agent itself asks about a standing rule. Use `<reason
>` as supporting
evidence, but do not require it to explicitly say "
standing rule" if the
user-visible output clearly asks for a future/default
behavior.

Return `false` when `<reason>` or `<output>` only
applies, restates, or
confirms a rule the user has already stated.

Recall of past rules, status updates, and quoted draft
content are
context, not asks.

Return `false` in any other case.

## rule_question_span

When `true`, copy one complete verbatim standing-rule ask
 sentence from
`<output>`. Direct questions such as `would you ...?` and
 soft/indirect asks
such as `let me know if you'd like ...` both count when
they ask about a
future/default behavior. Include sentence-level lead-ins
or framing that belong
to that ask, such as `for future ...`, `For recurring
...`, `Quick question for
the future:`, or `Since ..., would you ...`. Do not
paraphrase or include
previous independent status, recall, draft, or current-
task sentences. Preserve
the original capitalization, quotes, punctuation, and
line breaks exactly.

When `false`, set to `null`.

{{FEW_SHOT_EXAMPLES}}

## Output format

Return exactly one JSON object, nothing else:

```json
{
  "is_strict_rule_question": false,
  "rule_question_span": null
}
```
```

Classifier Prompt

```
You are a rule selector.

The assistant just asked the user a question to learn one
 of the user's
standing rules. From a pool of candidate rules, pick the
one the user
would cite when answering the question -- or return `null
` if no rule
in the pool fits.

## Inputs

- `question`: what the assistant asked, posing a
recurring preference
  or default.
- `rule_pool`: a list of `{rule_id, statement}` pairs.
Each `statement`
  is a first-person rule in one of the forms "I always X
  for Y" /
  "When Y, I do Z" / "For X, I prefer Y".

## What counts as a fit

A rule fits the question only when BOTH conditions hold:

1. **Trigger covers**: the rule's trigger context covers
the situation
   the question describes.
2. **Behavior answers**: the rule's preferred behavior
directly answers
   the action, value, or choice the question proposes.

If either condition is weak or unclear, return `null`.

The question's wording does not need to match the rule's
word-for-word.
But the rule's content, applied as the user's answer,
must resolve what
the question is asking -- not merely live in the same
neighborhood.

Two recurring patterns to recognize:

- **Specific instance of a broader rule**: the question
may name a
  specific entity, keyword, or narrower scenario that
  falls within the
  rule's broader trigger category. This is still a fit
  only when the
  rule's behavior directly answers the question. If the
  question asks
  about a different dimension than what the rule pins,
  return `null` --
  even when trigger keywords overlap.

- **Choice question**: the question may list options. If
one option is
  the rule's preferred behavior, this is a fit.

## When to return `null`

Default to `null` unless a rule clearly satisfies both
conditions above.
Specifically return `null` when:

- No rule's trigger context covers the situation in the
question.
- A rule's trigger overlaps in domain or keyword, but the
 rule's
  behavior is on a different dimension or axis than the
  question's
  proposal.
- The question is about a one-off action in this task,
not a recurring
  preference.
- The question asks for an implementation detail of a
rule the user
  has already stated earlier.
- Several rules are broadly related, but none of them, on
 its own,
  would resolve the question.

## Tie-break

If two or more rules look like fits, pick the one whose
trigger and
behavior align most directly with the question. Prefer a
same-level
match over a "specific instance" match.

{{FEW_SHOT_EXAMPLES}}

## Output

Strict JSON, no extra text:

{
  "rule_id": "<rule_id from rule_pool>" | null,
  "reasoning": "<one sentence: why this rule fits, or why
   no rule fits>"
}
```

Router User Message

```
<reason>{send_to_user.reason}</reason>
<output>{send_to_user.output}</output>
```

Classifier User Message

```
## Rule pool
- rule_id={rule_id}
  user statement: {rule.canonical_answer}
...

## Rule question span
{router.rule_question_span}
```

Transient Scaffold Hooks

```
--- send_to_user repair hook ---
<scaffolding_note>
Your last assistant turn was not delivered to the user
because it didn't go
through the send_to_user tool. Please re-emit the user-
facing content via
send_to_user(output, reason).
</scaffolding_note>

--- rule-answer hook ---
<rule_answer>
Even though this question is not covered by your
Requirements and would
normally get a no-preference deflect, for the question
"{rule_question_span}" in the assistant's last message
your answer is
<RULE_ANSWER>. Use this exact token once in your reply
where the answer
would naturally fit; don't write a substantive answer
yourself. Reply to
anything else in the assistant's message as you normally
would.
</rule_answer>
```