LAMP: IDENTIFYING LOAD-BEARING ASSUMPTIONS FOR OUTCOME-SENSITIVE CLARIFICATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Language agents routinely act on underspecified instructions by silently committing to one interpretation. Yet ambiguity alone does not determine whether clarification is necessary: different valid readings can lead to the same task outcome. We call an unresolved assumption *load-bearing* when resolving it differently can materially change the task outcome. Good clarification thus requires coverage (asking about load-bearing assumptions) and selectivity (not asking unnecessarily). We introduce LAMP (*Load-bearing AssuMPtion*), a clarification framework that detects unresolved assumptions, enumerates their readings, and decides which are load-bearing, using structured reasoning or counterfactual execution. Since ambiguity annotations do not reveal whether the outcome materially changes, we construct *execution-grounded* evaluation cases: starting from a task that a fixed agent solves correctly, we remove one specification detail and rerun the agent; an observed outcome change yields a load-bearing evaluation target, while outcome preservation yields a negative control for the tested context. On text-to-SQL (BIRD), LAMP recovers 95% of load-bearing evaluation targets, versus 72% for the strongest prior method, and counterfactual execution makes clarification more selective at nearly the same coverage. The formulation transfers to software engineering (SWE-bench) without domain-specific adaptation, and the clarification policy can be distilled into a 7B model while keeping the execution agent fixed. Together, these results suggest that agents should not request clarification whenever an instruction is ambiguous, but when resolving the ambiguity differently can change the task outcome.

1 INTRODUCTION

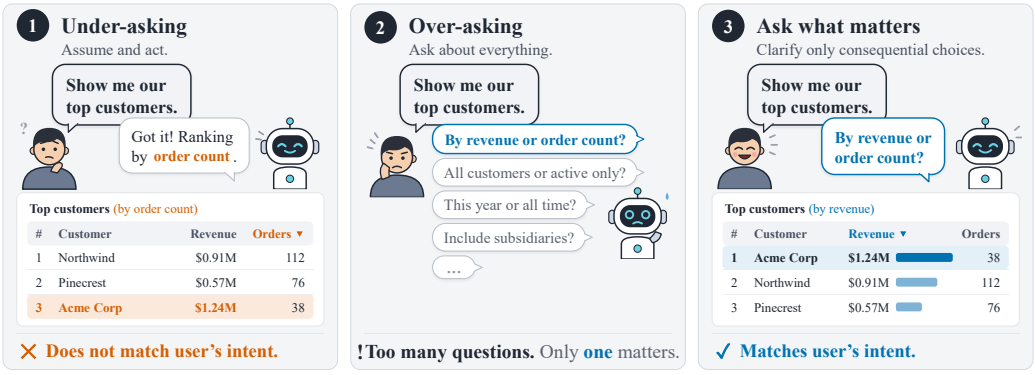


Figure 1: **Clarification requires both coverage and selectivity.** Under-asking *(1)* can miss an unresolved assumption that changes the task outcome, while over-asking *(2)* can burden the user with questions whose answers leave the outcome unchanged. The goal *(3)* is to ask only about assumptions that can change the outcome.

Language agents carry out tasks from natural-language instructions, from writing database queries to editing code to orchestrating tools (Li et al., 2023; Schick et al., 2023; Yang et al., 2024). Such instructions are rarely fully specified: a request to "Show me our top customers" leaves open how to rank customers, which customers are in scope, and over what time period. The agent must then either

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

make assumptions, which may not match the user's intent, or ask for clarification, which burdens the user with questions that may not affect the result. Figure 1 illustrates this trade-off.

Prior work has studied ambiguous and underspecified language in question answering (Min et al., 2020; Kim et al., 2024; Zhang et al., 2024), examined task-level underspecification (Tamkin et al., 2023), and developed methods for generating clarifying questions in interactive settings (Zhang & Choi, 2025; Zhang et al., 2025; Chen et al., 2025). These efforts address whether an instruction *is* ambiguous, but not whether the ambiguity *matters*. In our example, ranking by revenue instead of order count changes which customers are shown, whereas restricting the query to active customers changes nothing if all top-ranked customers are active in the current data. The relevant question is therefore whether resolving the ambiguity differently would change the task outcome.

We formalize this distinction through **load-bearing assumptions**. An unresolved assumption is load-bearing when alternative valid readings can lead to materially different task outcomes, holding the remaining assumptions fixed. This turns clarification into an outcome-sensitive decision: the goal is neither to ask about every plausible ambiguity nor simply to minimize the number of questions, but to clarify the assumptions that can change the task outcome. We introduce LAMP (*Load-bearing AssuMPtion*), a framework for selective clarification. LAMP identifies unresolved assumptions, makes their plausible readings explicit, and assesses whether resolving them differently changes the task outcome, either by model-based reasoning or by counterfactual execution, which runs the task under alternative readings and compares the results. Only unresolved assumptions predicted to be load-bearing are passed to the user.

Evaluating whether a clarification method asks the right questions requires knowing whether competing readings actually lead to different outcomes, which ambiguity-focused benchmarks (Min et al., 2020; Zhang et al., 2024) do not directly provide. We therefore construct *execution-grounded* evaluation cases: starting from tasks that a fixed execution agent solves correctly, we remove one specification detail and rerun the same agent. If the outcome changes, the assumption made unresolved by the removed detail is load-bearing at the tested realization and serves as the evaluation target a clarification method should recover. If the outcome remains unchanged, the detail did not affect the result, and the case tests whether a method avoids unnecessary clarification. On BIRD SQL (Li et al., 2023), LAMP recovers 95% of load-bearing evaluation targets and reaches 81% execution accuracy, compared with 61% for the strongest existing baseline (Vijayvargiya et al., 2026). Counterfactual execution makes clarification more selective at nearly the same recall, and both patterns hold with a different clarification model. The same formulation also transfers to SWE-bench (Jimenez et al., 2024) without domain-specific adaptation, and the clarification policy can be learned by a 7B open-weight model with the execution agent kept fixed.

In summary, our contributions are:

- **Criterion.** We formalize load-bearing assumptions as a criterion for when clarification is needed and show that, under simplifying assumptions, these assumptions suffice to determine the task outcome.
- **Framework.** LAMP identifies unresolved assumptions, enumerates their readings, and decides which are load-bearing, by structured reasoning or counterfactual execution.
- **Evaluation.** We construct execution-grounded load-bearing cases and outcome-preserving negative controls from observed execution.
- **Experiments.** We evaluate LAMP on text-to-SQL (BIRD) and software engineering (SWE-bench) tasks, and show that its clarification policy can be learned by a 7B model.

2 Related Work

Ambiguity and underspecification in language. Ambiguity has long been recognized as a central difficulty in natural-language specification, for example in requirements engineering, where lexical, referential, and structural ambiguities can change how a request is interpreted (Berry et al., 2003; Berry & Kamsties, 2004). With language models, ambiguity has been studied in open-domain question answering (Min et al., 2020), explicit ambiguity handling (Kim et al., 2024), and task-level underspecification (Tamkin et al., 2023). These works ask whether an instruction admits multiple plausible interpretations, whereas we ask whether changing among them can change the task outcome.

Clarification and selective interaction. Prior work studies when and what language models should ask. CLAMBER benchmarks identification and clarification of ambiguous information needs (Zhang et al., 2024), while subsequent work learns clarification behavior from interaction (Chen et al., 2025; Zhang & Choi, 2025; Zhang et al., 2025). Other approaches use model uncertainty to trigger clarification (Testoni & Fernández, 2024) or ask for help in language-model planning (Ren et al., 2023). Decision-theoretic approaches rank questions by expected value of information (Rao & Daumé III, 2018), or explicitly balance the expected utility of clarification against interaction cost (Dong et al., 2026; Suri et al., 2026). LAMP instead asks whether changing the reading of a particular unresolved assumption can materially change the task outcome while the remaining assumptions are held fixed.

Clarification in executable tasks. AmbiSQL uses a domain-specific ambiguity taxonomy for interactive ambiguity resolution in text-to-SQL (Ding et al., 2026), whereas Ambig-SWE studies underspecified software-engineering tasks (Vijayvargiya et al., 2026). ClarifyGPT detects ambiguous code-generation requirements through behavioral inconsistency among sampled programs and then generates targeted clarification questions (Mu et al., 2024). Learning to Ask similarly studies clarification under incomplete instructions for tool-using agents (Wang et al., 2025). In contrast, LAMP explicitly represents unresolved assumptions and their alternative readings, and evaluates whether changing a reading can alter the task outcome. Its execution-grounded evaluation likewise derives load-bearing targets and outcome-preserving negative controls from observed execution.

3 PRELIMINARIES

Setup. We model the execution agent as π_{exec}, which receives a natural-language instruction $u \in \mathcal{U}$ together with task context $c \in \mathcal{C}$ (e.g., a database schema, a codebase, or conversation history) and produces a task outcome $y \in \mathcal{Y}$. Two outcomes y and y' are *materially equivalent*, written $y \equiv y'$, when they agree under a specified task-level comparison (e.g., two SQL queries that return identical result sets, or two code patches that pass the same test suite). We assume $\equiv$ is an equivalence relation over $\mathcal{Y}$. For the formal analysis, we fix the instruction, context, agent, execution procedure, and any execution randomness.

Unresolved assumptions and readings. Natural-language instructions may leave important aspects unresolved (Berry & Kamsties, 2004; Min et al., 2020). We model these as K unresolved assumptions $A_1, \ldots, A_K$, where each assumption A_j has a finite set of valid readings $\mathcal{A}_j$, with $|\mathcal{A}_j| \geq 2$. A reading is a concrete resolution of the corresponding assumption. A *realization*

$$\mathbf{a} = (a_1, \ldots, a_K) \in \mathcal{A} := \prod_{j=1}^{K} \mathcal{A}_j$$

selects one reading for every unresolved assumption.

For the formal analysis, we treat the decomposition into $A_1, \ldots, A_K$ as given, and it need not be unique. We assume that all realizations in $\mathcal{A}$ are jointly valid. This does not assume that the effects of different assumptions are independent: the outcome may depend on interactions among their readings.

We write $y(\mathbf{a})$ for the outcome produced by π_{exec} when the readings in $\mathbf{a}$ are explicitly specified. Let $\mathbf{a}^* \in \mathcal{A}$ denote the user's intended realization and $y^* \in \mathcal{Y}$ the intended outcome; we assume

$$y(\mathbf{a}^*) \equiv y^*.$$

To connect this full-realization model to clarification, we further assume that execution under any partial specification is materially equivalent to $y(\mathbf{a})$ for some valid completion $\mathbf{a} \in \mathcal{A}$ consistent with the specified readings.

Load-bearing assumptions. The effect of an assumption can depend on how the remaining assumptions are realized. We therefore first define load-bearingness relative to a particular realization.

Definition 1 (Load-bearing at a realization)**.** *An assumption A_j is* load-bearing at realization $\mathbf{a}$ *if there exists an alternative reading $a'_j \in \mathcal{A}_j \setminus \{a_j\}$ such that*

$$y(a'_j, \mathbf{a}_{-j}) \not\equiv y(\mathbf{a}),$$

where $\mathbf{a}_{-j}$ denotes the readings of all assumptions other than A_j.

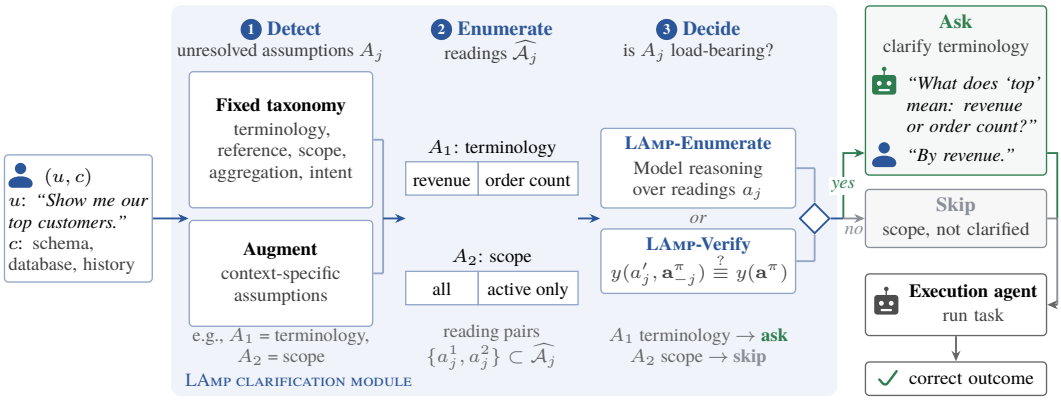


Figure 2: **Overview of LAMP.** Given an instruction u and task context c, LAMP detects unresolved assumptions A_j, enumerates their readings $\widehat{\mathcal{A}}_j$, and assesses whether each assumption is load-bearing. LAMP-Enumerate predicts load-bearingness from the enumerated readings, whereas LAMP-Verify tests alternative readings through execution and compares the resulting outcomes. Load-bearing assumptions are clarified with the user, and the response is incorporated into the instruction before execution by π_{exec}.

An assumption A_j is a *load-bearing assumption* if it is load-bearing at at least one realization $\mathbf{a} \in \mathcal{A}$. If no such realization exists, changing the reading of A_j never changes the task outcome while the remaining readings are held fixed. Thus, a single outcome-changing realization is sufficient to establish load-bearingness, whereas observing no outcome change at one realization does not establish that the assumption is non-load-bearing in general.

Let

$$L = \{j \in \{1, \ldots, K\} : A_j \text{ is load-bearing}\}$$

denote the indices of the load-bearing assumptions.

Proposition 1 (Outcome determination by load-bearing assumptions)**.** *For any two realizations* $\mathbf{a}, \mathbf{b} \in \mathcal{A}$*,*

$$a_j = b_j \quad \forall j \in L \quad \Longrightarrow \quad y(\mathbf{a}) \equiv y(\mathbf{b}).$$

Proof. Let $j_1, \ldots, j_m$ be the indices outside L on which $\mathbf{a}$ and $\mathbf{b}$ differ. Change these readings one at a time from $\mathbf{a}$ to $\mathbf{b}$. Every intermediate realization lies in $\mathcal{A}$, and each step changes only an assumption that is not load-bearing, so the task outcome remains materially equivalent. By transitivity of $\equiv$, $y(\mathbf{a}) \equiv y(\mathbf{b})$. □

Consequently, if clarification supplies the user's intended readings for all load-bearing assumptions, then any valid completion consistent with those readings produces an outcome materially equivalent to y^*. A clarification method should therefore recover the load-bearing assumptions (*coverage*) while leaving the rest unresolved (*selectivity*), as Proposition 1 guarantees that non-load-bearing assumptions do not affect the outcome.

4 Proposed Approach: LAMP

Overview. Proposition 1 establishes that matching the user's intended readings on all load-bearing assumptions is sufficient for outcome agreement. This motivates a clarification policy that identifies assumptions whose alternative readings can materially change the task outcome.

We introduce LAMP (*Load-bearing AssuMPtion*), which identifies load-bearing assumptions through three operations. It detects unresolved assumptions in the instruction, enumerates the alternative valid readings of each assumption, and decides whether the assumption is load-bearing. Detection and enumeration produce a set of candidate assumptions, each with its enumerated readings. The final decision selects the assumptions to be resolved through clarification with the user (Figure 2).

Detecting unresolved assumptions. Given a user instruction u and context c, we first prompt a language model π_{clar} to identify the unresolved assumptions $A_1, \ldots, A_K$ that the instruction leaves

open. This proceeds in two stages. Motivated by the ambiguity literature in requirements engineering (Berry et al., 2003; Berry & Kamsties, 2004), in the first stage we provide π_{clar} with a fixed taxonomy of five sources of underspecification: terminology, reference, scope, aggregation, and intent. The model identifies unresolved assumptions of each kind, such as an undefined term (terminology) or an unstated selection boundary (scope). For instance, in "Show me our top customers", this surfaces the ranking criterion and the customer population as candidate unresolved assumptions. This fixed taxonomy covers ambiguity types that recur across tasks and can be specified beforehand. However, since a finite taxonomy can miss assumptions specific to an instruction and its context, in the second stage we prompt π_{clar} to recover context-specific assumptions and, optionally, categories beyond the taxonomy. Appendix B.5 shows that this augmentation helps recover load-bearing assumptions that the fixed taxonomy alone does not detect.

Enumerating alternative readings. For each identified candidate assumption A_j, we prompt π_{clar} to construct a finite set of alternative readings compatible with (u, c):

$$\widehat{\mathcal{A}}_j = \{a_j^{(1)}, \ldots, a_j^{(m_j)}\}. \tag{1}$$

Each reading specifies a distinct resolution of the same assumption. For example, the ranking criterion in "Show me our top customers" may be resolved as ordering by revenue or by order count. The enumerated set $\widehat{\mathcal{A}}_j$ approximates the full reading set $\mathcal{A}_j$ and need not be exhaustive, since identifying A_j as load-bearing requires only two readings that produce materially different outcomes. Incomplete enumeration may in principle reduce coverage if such a reading is missed. However, in Section 6, we observe that LAMP recovers 95% of the load-bearing assumptions, indicating that enumeration rarely omits the readings that matter. We retain candidate assumptions with at least two valid enumerated readings. Validation and consolidation details are given in Appendix B.

Evaluating load-bearingness. For each retained assumption, LAMP assesses whether changing its reading can materially alter the task outcome. The formal criterion holds the remaining assumptions fixed. LAMP-Enumerate estimates this effect from model judgment, whereas LAMP-Verify compares observed outcomes from executions conditioned on the enumerated readings when a task-level equivalence test is available.

LAMP-Enumerate. For each retained candidate assumption A_j and its readings $\widehat{\mathcal{A}}_j$, π_{clar} predicts whether any two enumerated readings can produce materially different outcomes while the remaining assumptions are held fixed. This prediction approximates the load-bearing check without executing the alternatives.

LAMP-Verify. When controlled execution is available, LAMP-Verify replaces the model-based judgment with observed execution evidence. We first execute the original instruction with π_{exec}. By the partial-specification assumption in Section 3, let $\mathbf{a}^{\pi} \in \mathcal{A}$ be a realization whose outcome is materially equivalent to this direct execution.

For each reading $a'_j \in \widehat{\mathcal{A}}_j$, we rerun π_{exec} with A_j resolved as a'_j while holding the remaining readings fixed at $\mathbf{a}^{\pi}_{-j}$, and compare

$$y(a'_j, \mathbf{a}^{\pi}_{-j}) \stackrel{?}{\equiv} y(\mathbf{a}^{\pi}). \tag{2}$$

If any $a'_j \in \widehat{\mathcal{A}}_j$ yields an outcome that differs materially from $y(\mathbf{a}^{\pi})$, A_j is load-bearing at $\mathbf{a}^{\pi}$ and is selected for clarification. If all $a'_j \in \widehat{\mathcal{A}}_j$ yield an outcome equivalent to $y(\mathbf{a}^{\pi})$, we conclude that A_j is not load-bearing at $\mathbf{a}^{\pi}$. Executions that do not permit a meaningful comparison are left undetermined (Appendix A.2).

The two variants address the coverage and selectivity requirements differently. LAMP-Enumerate favors coverage by predicting load-bearingness from the readings alone, while LAMP-Verify improves selectivity by retaining only assumptions whose alternative readings are observed to change the executed outcome. Assumptions selected as load-bearing are converted into clarification questions and addressed to the user. The responses are then incorporated into the task specification before execution by π_{exec}. If no assumption is selected, π_{exec} executes the original instruction without clarification.

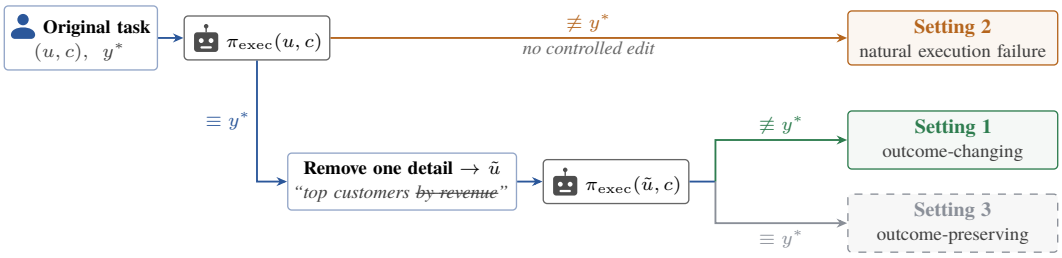


Figure 3: **Execution-grounded evaluation construction.** The execution agent π_{exec} is first run on the original task (u, c). If it does not produce the intended outcome y^*, the task is retained as a natural execution failure (Setting 2) without any controlled edit. If it succeeds, we remove one specification detail and rerun the same agent on the edited instruction $\tilde{u}$. An outcome that no longer matches y^* identifies the removed detail as load-bearing (Setting 1). An outcome that remains equivalent to y^* across repeated executions provides an evaluation control where clarification is unnecessary (Setting 3).

5 Constructing Execution-Grounded Evaluation Cases

Evaluating outcome-sensitive clarification requires knowing which unresolved assumptions actually change the task outcome. Standard ambiguity benchmarks annotate whether an instruction is ambiguous, but not whether that ambiguity matters (Min et al., 2020; Zhang et al., 2024). We therefore construct *execution-grounded* evaluation cases using observed execution outcomes.

We start from an instruction u, task context c, and intended outcome y^*, and execute the task with the fixed execution agent π_{exec}. As illustrated in Figure 3, tasks that π_{exec} already fails on are retained as natural execution failures. For tasks it solves correctly, we remove one specification detail from the instruction to obtain an underspecified variant $\tilde{u}$, and rerun the same execution agent. This produces three complementary evaluation sets.

Outcome-changing edits (Setting 1). If the agent's outcome on the edited instruction does not match y^*, then the assumption made unresolved by the removed detail is load-bearing at the tested realization and serves as the evaluation target. A clarification method should recover this assumption. Recovery is measured through recall, and downstream execution measures whether clarification restores the intended outcome.

Outcome-preserving edits (Setting 3). If the agent's outcome on the edited instruction still matches y^*, then the removed detail did not affect the result in the tested context. These cases serve as outcome-preserving negative controls: a selective clarification method should avoid unnecessary clarification when the tested omission leaves the outcome unchanged. This does not imply that the corresponding assumption is non-load-bearing in general, since its effect may depend on how other assumptions are realized. Question counts on this cohort measure how often a method triggers clarification in these outcome-preserving cases.

Natural execution failures (Setting 2). Tasks on which π_{exec} does not solve the original instruction correctly are retained without editing. We do not introduce any controlled underspecification for these tasks. These tasks test whether clarification can improve naturally occurring failures, without assuming that those failures arise from underspecification.

The controlled edits are constructed independently of LAMP's seed taxonomy, so the evaluation is not built around the ambiguity categories used by the method being tested. Evaluation targets are determined from observed execution rather than from LAMP's predictions. Appendix C provides the full edit-construction, grounding, repeated-execution, and cohort-assignment details.

6 Experiments

A clarification framework must not only recover load-bearing assumptions but also control how often it asks the user for input. In the experiments below, we evaluate these two requirements separately and examine whether recovering more load-bearing assumptions translates into improved execution accuracy.

Method	Target Rec. ↑	Target Prec. ↑	Exec Acc. ↑	n_q ↓	Cost ↓
External baselines					
Ambig-SWE (Vijayvargiya et al., 2026)	0.72	0.28	0.61±0.006	2.58	0.03
AmbiSQL (Ding et al., 2026)	0.68	0.43	0.58±0.006	1.65	0.10
CLAMBER (Zhang et al., 2024)	0.58	0.44	0.53±0.015	1.34	0.04
Simple load-bearing prompts					
LB prompt (zero-shot)	0.69	0.13	0.62±0.002	5.45	0.04
LB prompt (few-shot)	0.70	0.18	0.62±0.007	3.87	0.03
LAMP family					
LAMP-OneShot	0.86	0.23	0.71±0.008	3.86	0.04
LAMP-Enumerate	0.95	0.40	0.81±0.004	3.02	0.18
LAMP-Verify	0.94	0.42	0.77±0.002	2.64	0.26

Table 1: **Outcome-changing edits on BIRD (Setting 1, N=196).** Each instance contains at least one load-bearing target. Target Recall and Precision are computed from the frozen clarification outputs; Exec Acc. is mean±std over three downstream executions with these clarification decisions held fixed.

6.1 Experimental setup

Execution-grounded evaluation. We begin from 500 BIRD mini-dev SQL instances (Li et al., 2023) and apply the execution-grounded construction from Section 5. Setting 1 contains outcome-changing edits for which the removed assumption serves as a load-bearing evaluation target. Setting 2 contains naturally occurring failures of the execution agent, for which no evaluation target is available. Setting 3 contains outcome-preserving edits and serves as a negative control for unnecessary clarification (Appendix D). The main-study cohorts contain N=196 Setting 1 and N=240 Setting 2 instances; Setting 3 is evaluated on N=50 held-out instances.

Implementation details. For all evaluations, we use Claude Opus 4.8 as the clarification model π_{clar}, and Claude Sonnet 4.6 as the execution agent π_{exec} unless stated otherwise. To test transfer across backbones, we repeat Setting 1 with Terra (OpenAI API model identifier `gpt-5.6-terra`) as π_{clar}. For question-generation and simulating user responses, we use Sonnet 4.6. All prompted methods use the same question generator, simulated user, execution agent, and task-level outcome comparison; only the method that selects assumptions for clarification varies. Selected assumptions are converted into questions, and the responses are supplied to π_{exec} before final execution. The LAMP prompt templates and baseline protocols are described in Appendices B.6 and B.7.

Baselines. We compare LAMP against three ambiguity-based clarification methods: Ambig-SWE (Vijayvargiya et al., 2026), AmbiSQL (Ding et al., 2026), and CLAMBER (Zhang et al., 2024). These baselines test whether identifying ambiguity alone is sufficient to recover assumptions that materially affect the outcome. We additionally include comparison with zero- and few-shot prompts that directly identify load-bearing assumptions without the separate detection and enumeration steps. As an ablation, we perform detection, enumeration, and load-bearing evaluation in a single call to π_{clar} (we refer to this as LAMP-OneShot), isolating the contribution of separating these steps.

Evaluation Metrics. Let $\mathcal{L}^*$ be the annotated target assumptions and $\widehat{\mathcal{L}}$ the assumptions selected for clarification. We report **Target Recall**, the fraction of targets recovered; **Target Precision**, the fraction of selected assumptions that match a target semantically; **Exec Acc.**, the fraction of final outcomes satisfying $y \equiv y^*$; n_q, the mean number of clarification questions per instance; and Cost, the average API cost per instance in USD. For natural execution failures, which have no annotated targets, we report only Exec Acc. and n_q. We provide more details in Appendix B.8.

6.2 Main Evaluation Results

Identifying load-bearing assumptions requires more than detecting ambiguity. Table 1 reports results on outcome-changing edits, where each instance contains at least one load-bearing assumption that materially changes the outcome. We make four observations: *(1)* Asking the model to identify load-bearing assumptions directly is insufficient: the zero-shot and few-shot prompts recover only 69–70% of targets, below Ambig-SWE's 72%, despite asking as many or more questions. *(2)* Recovery improves when the method first makes unresolved assumptions and their readings explicit. At an almost identical question count, LAMP-OneShot recovers 86% of targets against 70% for the few-shot

Method	Exec Acc. ↑	n_q ↓	Cost ↓
External baselines			
Ambig-SWE (Vijayvargiya et al., 2026)	0.51	2.63	0.03
AmbiSQL (Ding et al., 2026)	0.33	1.62	0.15
CLAMBER (Zhang et al., 2024)	0.30	1.36	0.04
Simple load-bearing prompts			
LB prompt (zero-shot)	0.47	5.54	0.04
LB prompt (few-shot)	0.51	4.22	0.04
LAMP family			
LAMP-OneShot	0.47	4.25	0.05
LAMP-Enumerate	0.53	3.42	0.18
LAMP-Verify	0.55	2.85	0.29

Table 2: **Natural execution failures on BIRD (Setting 2, N=240).** The agent fails the original task and no edit is introduced, so we report execution accuracy and clarification frequency only; Exec Acc. is from a single downstream execution due to API cost.

Method	Target Rec. ↑	Target Prec. ↑	Exec Acc. ↑	n_q ↓
External baselines				
Ambig-SWE (Vijayvargiya et al., 2026)	0.60	0.27	0.23±0.067	2.33
CLAMBER (Zhang et al., 2024)	0.10	0.43	0.04±0.019	0.23
Simple load-bearing prompts				
LB prompt (few-shot)	0.80	0.20	0.39±0.019	4.53
LAMP family				
LAMP-OneShot	0.80	0.27	0.37±0.058	3.57
LAMP-Enumerate	0.90	0.38	0.40±0.033	3.43

Table 3: **Natural information gaps on SWE-bench (N=30).** Each task becomes solvable only when a requirement absent from the issue is supplied; that requirement is the target to recover (construction in Appendix F). To limit the cost of repository-level execution, we omit AmbiSQL, which is specific to text-to-SQL, and the zero-shot load-bearing prompt, which is weaker than its few-shot variant on BIRD (Table 1).

prompt; decomposing detection, enumeration, and load-bearing assessment raises recovery to 95% and precision from 0.23 to 0.40. Much of this gain originates in candidate generation: taxonomy guidance raises the fraction of targets covered by the candidates from 0.85 to 0.92, and context-specific augmentation and extension raise it to 0.95 (Appendix B.5). *(3)* Recovering the right assumptions yields the correct answer: LAMP-Enumerate reaches 81% execution accuracy, against 61% for Ambig-SWE, while asking 22% fewer questions than LAMP-OneShot. *(4)* Counterfactual execution makes clarification more selective: LAMP-Verify retains 94% recall while raising precision to 0.42 and reducing n_q to 2.64. These findings transfer consistently to a second clarification backbone: with Terra as π_{clar}, LAMP-Enumerate raises Target Recall from 0.79 to 0.87 and Exec Acc. from 0.66 to 0.72 over LAMP-OneShot, and LAMP-Verify reduces n_q from 2.78 to 2.31 while retaining 0.83 recall (Appendix E).

Clarifying load-bearing assumptions helps even for questions the agent cannot originally answer. Table 2 reports results on natural execution failures, where no edit is introduced and a failure need not stem from underspecification. LAMP-Verify attains the highest execution accuracy (55% vs. 53% for LAMP-Enumerate and 51% for Ambig-SWE) while asking 17% fewer questions than LAMP-Enumerate. On outcome-preserving edits (Setting 3), where no clarification is needed, LAMP-Verify asks the fewest questions within the LAMP family (Appendix D).

6.3 Discussion

Evaluation on software engineering tasks. To test whether LAMP works beyond SQL, we curate 30 SWE-bench tasks that the execution agent fails to solve from the issue alone but solves once a requirement implied by the hidden tests is added (Appendix F). We annotate that requirement as the target; Table 3 reports the results. LAMP-Enumerate recovers 90% of targets, compared with 80% for few-shot prompting and 60% for Ambig-SWE, with higher precision (0.38) and fewer questions than the few-shot prompt (3.43 vs. 4.53), so the gain does not come from asking more. Execution gains are

Method	Target Rec. ↑	Target Prec. ↑	Exec Acc. ↑	n_q ↓
External baselines				
Ambig-SWE (Vijayvargiya et al., 2026)	0.57	0.35	0.57	1.60
AmbiSQL (Ding et al., 2026)	0.23	0.23	0.30	1.03
CLAMBER (Zhang et al., 2024)	0.17	0.56	0.27	0.30
Simple load-bearing prompts				
LB prompt (zero-shot)	0.50	0.20	0.47	2.60
LB prompt (few-shot)	0.33	0.13	0.37	2.50
LAMP family				
LAMP-OneShot	0.53	0.24	0.47	2.27
LAMP-SFT	0.67	0.46	0.63	1.53

Table 4: **Learning the clarification policy with a 7B backbone.** Held-out BIRD Setting 1 cohort (N=30). All methods use the same Qwen2.5-Coder-7B-Instruct clarification backbone where applicable and the same fixed Sonnet 4.6 downstream stack.

smaller than recovery gains, since the agent must still localize and implement the corresponding code change. We omit LAMP-Verify here because it would require repeated repository-level execution for every enumerated reading at inference time.

Guidance on choosing LAMP variants. Across Settings 1 and 2, LAMP-Verify asks 13 to 17% fewer questions than LAMP-Enumerate. However, its effect on execution accuracy varies: accuracy decreases from 81% to 77% on outcome-changing edits and increases from 53% to 55% on natural execution failures. We therefore favor LAMP-Verify when reducing user interaction is a priority and controlled execution of alternative readings is reliable and affordable. LAMP-Enumerate provides an alternative when repeated execution is costly or reliable outcome comparisons are unavailable.

Distilling the clarification policy into a smaller model. Running π_{clar} with a large model is costly at deployment, and prompting the multi-stage pipeline on a 7B model loses recall at each filtering stage (Appendix H). We therefore ask whether the clarification behavior can be learned instead. We fine-tune Qwen2.5-Coder-7B-Instruct with QLoRA to predict clarification decisions, keeping π_{exec} fixed, and refer to this policy as LAMP-SFT. We hold out 100 of the 500 BIRD questions and train on the remaining questions; Table 4 reports the 30 outcome-changing edits derived from the held-out questions, with split details and outcome-preserving results in Appendices G.1 and D. We use LAMP-Verify to label the training questions: assumptions it identifies as load-bearing provide positive examples, outcome-preserving edits provide negative examples, and the two are mixed 1:1. At inference time, the model sees only the instruction and schema and predicts whether to clarify, and if so, the unresolved assumptions and their readings (Appendix G). LAMP-SFT improves over the prompted LAMP-OneShot policy on the same 7B model across every metric: recall rises from 0.53 to 0.67, precision from 0.24 to 0.46, and execution accuracy from 0.47 to 0.63, while n_q drops from 2.27 to 1.53. The supervision source matters: with the same negatives, replacing LAMP positives with constructed outcome-changing edits lowers Target Recall from 0.70 to 0.43 (Appendix G.3).

7 Conclusion

We introduced *load-bearing assumptions* as a criterion for deciding which unresolved assumptions warrant clarification, LAMP for identifying them by model-based judgment or counterfactual execution, and execution-grounded evaluation cases that distinguish outcome-changing edits, outcome-preserving edits, and natural execution failures. Across BIRD and SWE-bench, explicitly identifying assumptions and their readings improves recovery of annotated targets and downstream execution. On BIRD, LAMP recovers more targets than direct load-bearing prompting, and counterfactual execution verification asks fewer questions while retaining similar target recall, though its effect on execution accuracy varies by setting. These patterns hold across two clarification backbones, and clarification decisions can be distilled into a smaller model. The central takeaway is: language agents should not ask whenever an instruction is ambiguous. Rather, they should ask when resolving the ambiguity differently can change the task outcome. We discuss limitations in Appendix H.

Ethics Statement

This work evaluates when language agents should request clarification. We use benchmark SQL tasks and software-engineering issues; no human participants are involved. User responses in our experiments are simulated, so the results do not establish how people would respond to clarification questions in deployment. Our criterion concerns material differences in task outcomes and should not be treated as a substitute for user preferences or domain-specific safety checks.

AI Use Statement

Language models are components of the methods and evaluations studied in this paper, including assumption identification, clarification question generation, simulated user responses, and task execution. We used LLM-based assistants for editing text and as coding assistants for routine engineering: experiment and evaluation scripts, the analysis scripts that aggregate logged runs into tables and figures, and code cleanup. The method, the experimental design, and the interpretation of results are the authors' own. The authors reviewed all text and code and take full responsibility for the content of the paper.

Reproducibility Statement

We specify the evaluation cohorts, model roles, outcome comparisons, and metrics in the main paper (Sections 5 and 6), and the appendix details cohort construction and repeated-execution procedures (Appendix C), method algorithms (Appendix B, Algorithm 1), prompt templates (Appendices B.6 and B.7), target matching (Appendix B.8), and fine-tuning hyperparameters (Appendix G.2). The supplementary material contains code for LAmp and all BIRD-SQL baselines, the shared evaluation pipeline (question generator, simulated user, execution agent, scorer, and metrics), and sample instances illustrating the data format; the BIRD databases are publicly available. Code for the SWE-bench experiments and the supervised fine-tuning pipeline, together with the full evaluation cohorts, will be released upon publication.

References

Daniel M. Berry and Erik Kamsties. Ambiguity in requirements specification. In Julio Cesar Sampaio do Prado Leite and Jorge Horacio Doorn (eds.), *Perspectives on Software Requirements*, pp. 7–44. Springer, Boston, MA, 2004. 2, 3, 5

Daniel M. Berry, Erik Kamsties, and Michael M. Krieger. From contract drafting to software specification: Linguistic sources of ambiguity—a handbook. https://cs.uwaterloo.ca/~dberry/handbook/ambiguityHandbook.pdf, 2003. Version 1.0. 2, 5

Maximillian Chen, Ruoxi Sun, Tomas Pfister, and Sercan Ö. Arık. Learning to clarify: Multi-turn conversations with action-based contrastive self-training. In *International Conference on Learning Representations (ICLR)*, 2025. 2, 3

Zhongjun Ding, Yin Lin, Tianjing Zeng, Rong Zhu, Bolin Ding, and Jingren Zhou. AmbiSQL: Interactive ambiguity detection and resolution for text-to-SQL. In *Companion of the ACM International Conference on Management of Data (SIGMOD), Demonstration Track*, pp. 26–29, 2026. 3, 7, 8, 9, 20, 21

Yijiang River Dong, Tiancheng Hu, Zheng Hui, Caiqi Zhang, Ivan Vulić, Andreea Bobu, and Nigel Collier. Value of information: A framework for human–agent communication. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 42879–42896, 2026. 3

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations (ICLR)*, 2024. 2

Hyuhng Joon Kim, Youna Kim, Cheonbok Park, Junyeob Kim, Choonghyun Park, Kang Min Yoo, Sang-goo Lee, and Taeuk Kim. Aligning language models to explicitly handle ambiguity. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1989–2007, 2024. 2

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. Can LLM already serve as a database interface? A BIg bench for large-scale database grounded text-to-SQLs. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2023. 1, 2, 7

Sewon Min, Julian Michael, Hannaneh Hajishirzi, and Luke Zettlemoyer. AmbigQA: Answering ambiguous open-domain questions. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 5783–5797, 2020. 2, 3, 6

Fangwen Mu, Lin Shi, Song Wang, Zhuohao Yu, Binquan Zhang, ChenXue Wang, Shichao Liu, and Qing Wang. ClarifyGPT: A framework for enhancing LLM-based code generation via requirements clarification. *Proceedings of the ACM on Software Engineering (FSE)*, 1(FSE):2332–2354, 2024. 3

Sudha Rao and Hal Daumé III. Learning to ask good questions: Ranking clarification questions using neural expected value of perfect information. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 2737–2746, 2018. 3

Allen Z. Ren, Anushri Dixit, Alexandra Bodrova, Sumeet Singh, Stephen Tu, Noah Brown, Peng Xu, Leila Takayama, Fei Xia, Jake Varley, Zhenjia Xu, Dorsa Sadigh, Andy Zeng, and Anirudha Majumdar. Robots that ask for help: Uncertainty alignment for large language model planners. In *Conference on Robot Learning (CoRL)*, pp. 661–682, 2023. 3

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. 1

Manan Suri, Puneet Mathur, Nedim Lipka, Franck Dernoncourt, Ryan A. Rossi, and Dinesh Manocha. Structured uncertainty guided clarification for LLM agents. In *Findings of the Association for Computational Linguistics (ACL Findings)*, pp. 40811–40838, 2026. 3

Alex Tamkin, Kunal Handa, Avash Shrestha, and Noah Goodman. Task ambiguity in humans and language models. In *International Conference on Learning Representations (ICLR)*, 2023. 2

Alberto Testoni and Raquel Fernández. Asking the right question at the right time: Human and model uncertainty guidance to ask clarification questions. In *Proceedings of the Conference of the European Chapter of the Association for Computational Linguistics (EACL)*, pp. 258–275, 2024. 3

Sanidhya Vijayvargiya, Xuhui Zhou, Akhila Yerukola, Maarten Sap, and Graham Neubig. Ambig-SWE: Interactive agents to overcome underspecificity in software engineering. In *International Conference on Learning Representations (ICLR)*, 2026. 2, 3, 7, 8, 9, 20, 21

Wenxuan Wang, Juluan Shi, Zixuan Ling, Yuk-Kit Chan, Chaozheng Wang, Cheryl Lee, Youliang Yuan, Jen-tse Huang, Wenxiang Jiao, and Michael R. Lyu. Learning to ask: When LLM agents meet unclear instruction. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 21773–21784, 2025. 3

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. 1

Michael J. Q. Zhang and Eunsol Choi. Clarify when necessary: Resolving ambiguity through interaction with LMs. In *Findings of the Association for Computational Linguistics (NAACL Findings)*, pp. 5541–5558, 2025. 2, 3

Michael J. Q. Zhang, W. Bradley Knox, and Eunsol Choi. Modeling future conversation turns to teach LLMs to ask clarifying questions. In *International Conference on Learning Representations (ICLR)*, 2025. 2, 3

Tong Zhang, Peixin Qin, Yang Deng, Chen Huang, Wenqiang Lei, Junhong Liu, Dingnan Jin, Hongru Liang, and Tat-Seng Chua. CLAMBER: A benchmark of identifying and clarifying ambiguous information needs in large language models. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 10746–10766, 2024. 2, 3, 6, 7, 8, 9, 20, 21

A Formal Details of the Load-Bearing Criterion

Section 3 defines load-bearingness over valid realizations of unresolved assumptions. We distinguish a controlled comparison that witnesses this formal criterion from the executions used by LAmp-Verify.

A.1 Controlled comparison

For a valid realization $\mathbf{a} \in \mathcal{A}$ and an alternative reading $a'_j \in \mathcal{A}_j \setminus \{a_j\}$, the formal comparison is

$$y(a'_j, \mathbf{a}_{-j}) \quad \text{versus} \quad y(\mathbf{a}). \tag{3}$$

The readings of all assumptions other than A_j are identical in these two realizations. If the outcomes differ materially, A_j is load-bearing at $\mathbf{a}$ and hence load-bearing under Definition 1. One such comparison suffices because the definition is existential over valid realizations.

A.2 Interpretation of execution evidence

As described in Section 4, let $\mathbf{a}^\pi \in \mathcal{A}$ be a realization whose outcome is materially equivalent to direct execution of the original instruction. For each enumerated reading $a'_j \in \widehat{\mathcal{A}}_j$, LAmp-Verify executes the instruction with A_j resolved as a'_j. Let $\mathbf{a}' \in \mathcal{A}$ be a realization consistent with that execution. The method compares $y(\mathbf{a}')$ with $y(\mathbf{a}^\pi)$ and also compares outcomes obtained under different enumerated readings.

These comparisons test whether the observed outcomes depend on the specified reading. They coincide with the controlled comparison in Eq. 3 when the other assumptions have the same readings across the compared executions. The implementation does not explicitly enforce this condition or identify the individual readings in $\mathbf{a}^\pi$.

A lack of observed difference applies to the tested readings and executions. In particular, the enumerated set $\widehat{\mathcal{A}}_j$ need not exhaust $\mathcal{A}_j$, and the effect of A_j may depend on the readings of other assumptions. Executions that cannot support a meaningful comparison receive an undetermined result rather than a conclusion about load-bearingness.

A.3 Relation to the clarification decision

Under a correctness-first policy with low interaction cost, observed material differences motivate clarifying A_j. When the tested executions permit a meaningful comparison and yield materially equivalent outcomes, clarification is not selected on that evidence. The decision may also depend on the cost of asking, the cost and reversibility of an incorrect outcome, and whether one response is safe under every valid reading. We leave these utility-sensitive extensions to future work.

B LAmp Algorithmic Details

This section provides implementation details omitted from the main method description, including candidate consolidation, reading validation, execution-based verification, and the full LAmp-Verify procedure. Code implementing LAmp, all BIRD-SQL baselines, and the shared evaluation pipeline is provided in the supplementary material.

B.1 Candidate identification

LAmp first identifies unresolved assumptions using the five-category taxonomy from Section 4: terminology, reference, scope, aggregation, and intent. Detected assumptions are anchored to the relevant part of the instruction when possible.

To improve coverage, we use two complementary expansion passes. *Augment* identifies assumptions that the agent must resolve to execute the task but that the instruction leaves unspecified, including assumptions that are not naturally expressed by a single ambiguous phrase. *Extend* allows additional ambiguity dimensions beyond the five-category seed taxonomy when needed.

Candidate generation	Candidate Rec. ↑	Avg. candidates / instance
No taxonomy	0.85	2.58
Taxonomy	0.92	4.08
Taxonomy + Augment	0.94	4.56
Taxonomy + Augment + Extend	0.95	5.66

Table 5: **Candidate-generation coverage ablation on BIRD Setting 1** (N=196). Each row progressively adds the candidate-identification components described in Appendix B.1.

The resulting candidate assumptions are passed to reading enumeration. Candidates corresponding to the same underlying assumption are consolidated before load-bearing assessment to avoid redundant clarification.

B.2 Reading enumeration

For each candidate assumption A_j, we prompt π_{clar} to enumerate a finite set of plausible readings $\widehat{\mathcal{A}}_j$. Each reading must be compatible with the instruction and task context and provide a concrete resolution of the unresolved assumption. Candidates with fewer than two valid readings are discarded.

The enumerated set need not be exhaustive: it provides the readings considered by the subsequent load-bearing assessment.

B.3 Deciding load-bearingness

Given a candidate assumption and its enumerated readings, LAmp determines whether alternative readings can materially change the task outcome. We consider two variants depending on whether controlled execution is available.

LAmp-Enumerate. Without executing the alternatives, π_{clar} judges whether the enumerated readings of a candidate assumption can lead to observably different task outcomes. The model receives the instruction together with the candidate assumption and its enumerated readings and returns a load-bearing or non-load-bearing judgment.

LAmp-Verify. When execution is available, we replace model-based judgment with observed execution evidence. We execute the original instruction once and reuse its outcome across candidate assumptions. For each retained assumption A_j, we then execute the instruction under every reading $a'_j \in \widehat{\mathcal{A}}_j$ and compare the resulting valid outcomes. If the executions do not permit a meaningful comparison, the candidate assumption is marked *Undetermined*. Algorithm 1 gives the full procedure.

Other LAmp variants. LAmp-Enumerate shares the candidate-identification and reading-enumeration stages with LAmp-Verify but makes the model-based load-bearing judgment above instead of executing readings. LAmp-OneShot performs candidate identification, reading enumeration, and load-bearing assessment jointly in a single model call.

B.4 Shared clarification and execution

Question generation is shared across all methods and is not part of LAmp. The selected load-bearing assumptions are passed to the same question generator, which consolidates assumptions referring to the same underlying assumption to avoid redundant clarification. The resulting user responses are incorporated into the original task specification before execution by π_{exec}.

B.5 Candidate-generation coverage ablation

The structured LAmp variants first identify candidate assumptions before assessing whether they are load-bearing. We isolate this candidate-identification stage on BIRD Setting 1 and measure whether the generated candidate set contains the target assumption before load-bearing assessment (Table 5).

Algorithm 1 LAMP-Verify

Require: instruction u, context c, clarification model π_{clar}, execution agent π_{exec}
Ensure: predicted load-bearing assumptions $\widehat{\mathcal{L}}$

identify candidate assumptions $\mathcal{C}$ using taxonomy-guided detection
augment $\mathcal{C}$ with context-specific assumptions
optionally extend beyond the seed taxonomy
for each candidate assumption $A_j \in \mathcal{C}$ **do**
    enumerate readings $\widehat{\mathcal{A}}_j$ with π_{clar}
    if $|\widehat{\mathcal{A}}_j| < 2$ **then**
        remove A_j from $\mathcal{C}$
split multi-anchor candidates and filter overlapping reading sets
execute the original instruction with π_{exec}
denote its outcome by $y(\mathbf{a}^{\pi})$, where $\mathbf{a}^{\pi}$ is a consistent realization
$\widehat{\mathcal{L}} \leftarrow \emptyset$
for each retained assumption $A_j \in \mathcal{C}$ **do**
    $\mathcal{Y}_j \leftarrow [y(\mathbf{a}^{\pi})]$
    for each reading $a'_j \in \widehat{\mathcal{A}}_j$ **do**
        execute with A_j resolved as a'_j
        let $\mathbf{a}'$ denote a realization consistent with this execution, with a'_j as its jth reading
        append $y(\mathbf{a}')$ to $\mathcal{Y}_j$
    apply execution-validity checks and applicable retries
    if fewer than two valid outcomes remain in $\mathcal{Y}_j$ **then**
        mark A_j as UNDETERMINED
    else if two valid outcomes in $\mathcal{Y}_j$ differ materially **then**
        $\widehat{\mathcal{L}} \leftarrow \widehat{\mathcal{L}} \cup \{A_j\}$
    else if the common outcome is degenerate **then**
        mark A_j as UNDETERMINED
    else
        record no observed load-bearing effect for A_j
return $\widehat{\mathcal{L}}$

Finding. Taxonomy guidance raises candidate recall from 0.85 to 0.92. Augment increases it further to 0.94, and allowing taxonomy extension reaches 0.95. This supports using the taxonomy as a seed for candidate discovery rather than as a closed set of ambiguity types.

B.6 PROMPT TEMPLATES

The boxes below summarize the operative instructions given to π_{clar}, using the assumption and reading terminology of Section 3. They describe the prompts rather than reproduce their full API strings. Detection, grounding, augmentation, extension, enumeration, and load-bearing judgment are separate calls.

Taxonomy-guided candidate detection

Input: Instruction u and task context c.
Instruction: Examine the task for unresolved assumptions concerning terminology, reference, scope, aggregation, and intent. For each category, identify every distinct site at which multiple valid readings remain. First check whether a qualifier in the same sentence already resolves the apparent ambiguity. Anchor each detected assumption to the relevant phrase or sentences and briefly explain the unresolved reading.
Output: Structured category labels, specification status, anchors, and explanations.

Grounding check

Input: Instruction u, a detected category, and the reason it was flagged.
Instruction: Determine whether the candidate assumption is grounded in the instruction. Return either an exact triggering phrase or the sentences that jointly give rise to an inferred unresolved assumption. Reject a candidate whose apparent ambiguity is resolved by the surrounding text.
Output: Grounding decision, grounding type, triggering text, and a brief reason.

Augment: execution-dependent assumptions

Input: Instruction u and task context c.
Instruction: Reason from the execution required to complete the task. Identify decisions the agent must resolve whose readings are not specified by the instruction, including assumptions without a single ambiguous phrase. State the concrete competing readings and why the task leaves them open. Exclude decisions already resolved by the task and process details that cannot affect the delivered outcome.
Output: Candidate assumptions, categories, competing readings, and explanations.

Extend: assumptions beyond the seed taxonomy

Input: Instruction u, task context c, and the five seed categories.
Instruction: Identify unresolved assumptions that do not fit terminology, reference, scope, aggregation, or intent. Include one only when two valid readings can produce observably different task outcomes. Give the additional category a short name, anchor it to the task, and explain the competing readings. Return an empty set if the seed categories cover the task.
Output: Additional category names, anchors, and explanations.

Reading enumeration

Input: Instruction u, task context c, and the candidate assumptions with their anchors.
Instruction: For each candidate assumption, enumerate distinct concrete readings that a competent domain practitioner would accept as faithful to the task. Resolve every supplied anchor; do not replace it with a broader, unrelated ambiguity. For scope assumptions, vary what is included while preserving the action applied to included items. Consider relevant ordering, threshold, reference-point, aggregation, and matching alternatives. Avoid paraphrases of the same reading.
Output: Up to ten readings per candidate, each with a short canonical value and an optional description.

LAMP-Enumerate load-bearing judgment

Input: Instruction u and the already identified candidate assumptions with their enumerated readings.
Instruction: For each candidate, judge whether adopting different listed valid readings would produce an observable difference in the delivered outcome. Use the supplied assumptions and readings; do not introduce new ones or execute the task.
Output: One load-bearing judgment and a brief reason for each candidate assumption.

LAMP-Verify: SQL execution prompts

The same SQL-writing system prompt is used for direct execution and for every reading-conditioned execution. It supplies the database schema, requests one SQL query, and requires SQL-only output.

Direct execution, user message: "Question: [original instruction]. Write the SQL query."

Execution under reading a'_j, user message: "Question: [original instruction]. Interpret the ambiguous term as: [a'_j]. Write the SQL query."

Each generated SQL query is executed against the database, and its result is used for the outcome comparisons. The direct execution does not identify the individual readings in $\mathbf{a}^{\pi}$; resolving A_j as a'_j does not explicitly fix the remaining assumptions.

B.7 Baseline prompts and evaluation protocol

Baseline prompts. The zero-shot load-bearing prompt provides the load-bearing definition and asks for unresolved assumptions and a judgment for each. The few-shot version adds two worked examples. For the prior-work baselines, we use their published or released prompt components in a single-call evaluation: Ambig-SWE's instruction is adapted to SQL, AmbiSQL uses wording from its released prompts, and CLAMBER uses its published taxonomy and examples. The evaluated direct load-bearing prompt templates are reproduced below.

LB zero-shot: system prompt

```
You are analyzing an underspecified task.

List the assumptions an agent would have to make that are NOT already
↪  specified in the task description. For each assumption, state
↪  whether it is load-bearing (a different valid reading would produce
↪  observably different output for an end user or downstream consumer)
↪  or not.

Respond in JSON:
{"assumptions": [{"assumption": "...", "load_bearing": true/false}]}
```

LB zero-shot: user prompt

```
Task description:
{query}

List the assumptions (not already specified) and mark each load-bearing
↪  or not.
```

LB few-shot: system prompt

```
You are analyzing an underspecified task.

List the assumptions an agent would have to make that are NOT already
↪  specified in the task description. For each assumption, state
↪  whether it is load-bearing (a different valid reading would produce
↪  observably different output for an end user or downstream consumer)
↪  or not.

Here are worked examples:

{examples}

Respond in JSON:
{"assumptions": [{"assumption": "...", "load_bearing": true/false}]}
```

LB few-shot: user prompt

```
Task description:
{query}
```

Here `{query}` is replaced by the evaluated instruction. The two examples inserted at `{examples}` concern reference and aggregation; their source questions are reserved from scored evaluation. Unlike LAMP-OneShot, these direct prompts do not request explicit reading enumeration.

Shared evaluation protocol. All baselines use the same question generator, simulated user, execution agent π_{exec}, and task-level outcome scorer. For the BIRD-SQL prompted comparison, the model used for baseline detection and LAMP's prompted stages is held fixed.

B.8 Target matching

Target Recall and Target Precision require deciding whether an assumption selected for clarification corresponds to an annotated target. We use an LLM judge that receives one selected assumption (with its readings, for methods that enumerate them) and the target, and decides whether both refer to the same unresolved decision, independently of wording. The judge is Claude Opus 4.8 at temperature 0 and is fixed across all tables, independently of the model that produced the predictions being scored, including the Terra and Qwen rows. The judge is not told which method produced the selected assumption. Target Recall is the fraction of targets matched by at least one selected assumption, and Target Precision is the fraction of selected assumptions that match a target.

C Execution-Grounded Evaluation Construction

This appendix details the construction procedure from Section 5. Cohort assignment is based on the fixed execution agent's observed outcomes, independently of LAmp's predictions.

C.1 Controlled edit construction

Let u be the original instruction, c its task context, y^* the gold outcome, and π_{exec} the fixed execution agent. We first execute the original task:

$$\hat{y}_0 = \pi_{\text{exec}}(u, c). \tag{4}$$

If $\hat{y}_0 \not\equiv y^*$, the agent already fails on the original task. We retain it as a Setting 2 natural execution failure and do not introduce a controlled edit.

If $\hat{y}_0 \equiv y^*$, a separate constructor ψ proposes an edit e that removes one specification detail. Applying the edit produces an underspecified instruction $\tilde{u}$. An original task may produce several edited variants; each resulting instruction contains one controlled edit. We execute each edited task:

$$\hat{y} = \pi_{\text{exec}}(\tilde{u}, c). \tag{5}$$

If this execution is valid and $\hat{y} \not\equiv y^*$, the edit produces a witnessed outcome change. We retain the instance in Setting 1, with the assumption made unresolved by e as its evaluation target.

If the edited task instead yields a valid outcome equivalent to y^*, it becomes a candidate for Setting 3. We retain it only after the repeated-execution check below. Setting 3 provides outcome-preserving controls for measuring unnecessary clarification under the tested execution protocol.

C.2 Taxonomy-independent grounding regimes

The controlled-edit constructor does not use LAmp's seed taxonomy. It produces edits with different forms of textual grounding:

- **Explicit:** the unresolved detail is localized to one textual span.
- **Multi-span:** two or more spans jointly express the unresolved detail.
- **Implicit:** the unresolved detail has no single clean verbatim anchor.

Although the constructor may be prompted toward a grounding regime, the recorded type is determined from the resulting edit. Grounding type is evaluation metadata and is not supplied to the methods under evaluation.

C.3 Execution stability and cohort assignment

An initial outcome-preserving execution alone does not establish a stable Setting 3 control. For each such candidate, we execute the edited instruction a further $k = 5$ times and require every run to produce a valid outcome materially equivalent to y^*.

The resulting assignments are:

Algorithm 2 Execution-grounded evaluation construction

Require: instruction u, context c, gold outcome y^*, execution agent π_{exec}, controlled-edit constructor ψ, repeat count $k = 5$
Ensure: execution-grounded evaluation cases $\mathcal{R}$

$\hat{y}_0 \leftarrow \pi_{\text{exec}}(u, c)$
if $\hat{y}_0 \not\equiv y^*$ **then**
  return $\{(u, \textsc{Setting2})\}$
$\mathcal{C} \leftarrow \textsc{ProposeControlledEdits}(\psi, u, y^*)$
$\mathcal{R} \leftarrow \emptyset$
for each candidate edit $e \in \mathcal{C}$ **do**
  $\tilde{u} \leftarrow \textsc{ApplyEdit}(u, e)$
  if $\textsc{Malformed}(\tilde{u})$ **then**
    continue
  $g \leftarrow \textsc{ClassifyGrounding}(e)$
  $\hat{y} \leftarrow \pi_{\text{exec}}(\tilde{u}, c)$
  if not $\textsc{Usable}(\hat{y})$ **then**
    continue
  if $\hat{y} \not\equiv y^*$ **then**
    $\mathcal{R} \leftarrow \mathcal{R} \cup \{(\tilde{u}, \textsc{Setting1}, e, g)\}$
    continue
  $\mathcal{Y} \leftarrow \emptyset$
  for $r = 1, \ldots, k$ **do**
    $\mathcal{Y} \leftarrow \mathcal{Y} \cup \{\pi_{\text{exec}}(\tilde{u}, c)\}$
  if every outcome in $\mathcal{Y}$ is usable and materially equivalent to y^* **then**
    $\mathcal{R} \leftarrow \mathcal{R} \cup \{(\tilde{u}, \textsc{Setting3}, g)\}$
return $\mathcal{R}$

- **Setting 1 / outcome-changing edit:** the initial valid execution of the edited instruction witnesses an outcome different from y^*.
- **Setting 2 / natural execution failure:** the original instruction fails before any controlled edit.
- **Setting 3 / outcome-preserving edit:** the initial execution and all five additional executions of the edited instruction yield valid outcomes equivalent to y^*.
- **Discarded edit:** the edited instruction is invalid, an execution is unusable, or the repeated outcomes do not meet the Setting 3 criterion.

Setting 3 establishes outcome preservation for the tested edit, context, and execution protocol; it does not establish that the unresolved assumption is non-load-bearing under every realization.

C.4 Full construction algorithm

Algorithm 2 summarizes the full construction procedure described above.

C.5 Independence from LAmp

Cohort assignment uses execution outcomes relative to y^*, rather than the assumption detector or load-bearing predictor being evaluated. In Setting 1, the target is the assumption made unresolved by the edit whose execution witnesses an outcome change. Setting 3 is defined by outcome preservation under the repeated execution check. Thus, LAmp is evaluated against cases and targets it did not generate.

This construction is an offline evaluation procedure. LAmp-Verify instead tests readings of candidate assumptions during inference to inform its clarification decision.

Method	Exec Acc. ↑	n_q ↓
External baselines		
Ambig-SWE (Vijayvargiya et al., 2026)	0.98	1.58
AmbiSQL (Ding et al., 2026)	0.96	0.92
CLAMBER (Zhang et al., 2024)	0.98	0.62
Simple load-bearing prompts		
LB prompt (zero-shot)	0.96	2.48
LB prompt (few-shot)	0.96	3.02
LAMP family		
LAMP-OneShot	1.00	2.16
LAMP-Enumerate	0.96	0.98
LAMP-Verify	0.98	0.84

Table 6: **Setting 3 outcome-preserving edits with a Qwen2.5-Coder-7B-Instruct clarification backbone (N=50).**

Method	Exec Acc. ↑	n_q ↓
External baselines		
Ambig-SWE (Vijayvargiya et al., 2026)	1.00	2.42
AmbiSQL (Ding et al., 2026)	0.98	1.26
CLAMBER (Zhang et al., 2024)	0.90	1.10
Simple load-bearing prompts		
LB prompt (zero-shot)	0.98	5.12
LB prompt (few-shot)	0.96	3.50
LAMP family		
LAMP-OneShot	0.94	3.70
LAMP-Enumerate	0.98	2.88
LAMP-Verify	1.00	2.16

Table 7: **Setting 3 outcome-preserving edits with a Claude Opus 4.8 clarification backbone (N=50).** All methods, including AmbiSQL, use the shared question generator.

D SETTING 3: OUTCOME-PRESERVING EDITS

The 50 held-out Setting 3 instances (Appendix G) contain edits whose outcomes remained equivalent to the intended outcome in the initial execution and five additional runs. We evaluate how often methods request clarification on these cases using n_q, and whether the task remains correctly solved using Exec Acc. Tables 6 and 7 report results with Qwen and Opus clarification backbones, respectively.

Findings. Within both backbones, LAMP-Verify asks fewer questions than LAMP-OneShot and LAMP-Enumerate. On the Opus run, it attains 1.00 execution accuracy with 2.16 questions per instance; Ambig-SWE also attains 1.00 accuracy, with 2.42 questions. All methods evaluated with both backbones ask fewer questions in the Qwen run. The remaining question counts, especially with Opus, show that outcome-preserving edits still trigger substantial clarification.

E CROSS-BACKBONE EVALUATION

We repeat the BIRD Setting 1 comparison with Terra (OpenAI API model identifier `gpt-5.6-terra`) as a second clarification backbone, using the same evaluation protocol and the same method definitions; all other components, including the target-matching judge (Appendix B.8), are unchanged. The qualitative pattern from the primary clarification backbone (Claude Opus 4.8) is preserved: explicit decomposition improves coverage over one-shot reasoning, while execution verification gives up a small amount of recall for greater selectivity.

Table 8 reports the results. Relative to LAMP-OneShot, LAMP-Enumerate raises Target Recall from 0.79 to 0.87 and Target Precision from 0.19 to 0.35 while reducing n_q from 4.34 to 2.78. LAMP-Verify further reduces n_q to 2.31 and raises precision to 0.38, with a smaller reduction in recall to 0.83. These results replicate the main coverage–selectivity finding under a second clarification backbone.

Method	Target Rec. ↑	Target Prec. ↑	Exec Acc. ↑	n_q ↓
External baselines				
Ambig-SWE (Vijayvargiya et al., 2026)	0.56	0.32	0.50	1.77
AmbiSQL (Ding et al., 2026)	0.66	0.49	0.59	1.35
CLAMBER (Zhang et al., 2024)	0.49	0.46	0.44	1.07
Simple load-bearing prompts				
LB prompt (zero-shot)	0.69	0.12	0.61	5.64
LB prompt (few-shot)	0.68	0.15	0.62	4.49
LAMP family				
LAMP-OneShot	0.79	0.19	0.66	4.34
LAMP-Enumerate	0.87	0.35	0.72	2.78
LAMP-Verify	0.83	0.38	0.67	2.31

Table 8: **Cross-backbone evaluation with Terra (`gpt-5.6-terra`) as the clarification backbone on BIRD Setting 1 (N=196).** LAMP-Enumerate achieves the highest Target Recall and execution accuracy, while LAMP-Verify trades a small amount of coverage for higher precision and fewer clarification questions.

F EXECUTION-GROUNDED SWE-BENCH INFORMATION GAPS

The main SWE-bench experiment uses naturally occurring information gaps rather than controlled edits. Starting from the full SWE-bench test split, we retain an instance only when a fixed repository-level executor fails on the verbatim issue in three consecutive runs and succeeds in three consecutive runs after the hidden test patch is appended. Instances that remain unsolved with the test information, or are already solved without it, are excluded. Missing assumptions used as evaluation targets are annotated after screening and before method evaluation, independently verified, and then blind reviewed. The final cohort contains N=30 instances spanning ten repositories.

The screening establishes that information omitted from the issue matters to the fixed executor and is sufficient to change its outcome. It does not rely on a controlled edit or on LAMP's taxonomy, so we refer to these cases as *execution-grounded natural information gaps*.

Repeated execution here is used only for offline cohort screening. We do not run LAMP-Verify on SWE-bench because doing so would require repeated repository-level execution for multiple alternative readings of every predicted assumption during inference, which is substantially more expensive and less reliable than this one-time screen.

G SMALL-MODEL TRAINING DETAILS

We train a smaller clarification policy with Qwen2.5-Coder-7B-Instruct while keeping the downstream question-generation and execution pipeline fixed. The policy receives only the natural-language question and database schema; it does not receive candidate assumptions, taxonomy labels, or outputs from the large-model LAMP pipeline at inference time. Its output is either NOASK or ASK together with one or more underspecified spans, each with candidate readings; below, NOASK and ASK denote the corresponding training examples.

G.1 SUPERVISION

We compare two sources of ASK supervision. *Outcome-changing-edit supervision* uses Setting 1 outcome-changing edits, where removing a specification detail changes the observed execution outcome; the removed detail provides the ASK target. *LAMP-derived supervision* instead uses assumptions identified as load-bearing by LAMP-Verify.

For NOASK supervision, we primarily use Setting 3 outcome-preserving edits. We additionally consider a larger paired-original NOASK pool as an auxiliary control. For some configurations, ASK examples are balanced across the ambiguity taxonomy before training. The final model additionally balances the effective ASK:NOASK training mixture to 1:1. The taxonomy is used only for sampling and is not provided to the model at inference time.

ASK source	ASK samp.	NOASK source	Train A/N	Rec. ↑	Prec. ↑	Exec. ↑	n_q ↓
Outcome-changing-edit ASK + paired-original NOASK							
OC-edit	natural	Paired	334/500	0.37	0.52	0.33	0.70
OC-edit	balanced	Paired	387/500	0.40	0.50	0.47	0.83
OC-edit	balanced	None	387/0	0.33	0.42	0.40	0.80
OC-edit	balanced	Paired	387/387	0.33	0.48	0.27	0.70
Outcome-changing-edit ASK + outcome-preserving NOASK							
OC-edit	natural	OP-edit	334/166	0.43	0.57	0.47	0.77
OC-edit	balanced	OP-edit	387/166	0.33	0.48	0.37	0.70
OC-edit	balanced	None	387/0	0.37	0.42	0.43	0.87
OC-edit	balanced	OP-edit	387/387	0.30	0.43	0.37	0.70
LAMP-derived ASK + outcome-preserving NOASK							
LAMP	natural	OP-edit	465/166	0.70	0.41	0.60	1.80
LAMP	balanced	OP-edit	684/166	0.73	0.35	0.60	2.17
LAMP	balanced	None	684/0	0.60	0.50	0.50	1.23
LAMP	balanced	OP-edit	684/684	0.67	0.46	0.63	1.53

Table 9: **Supervision ablations for LAMP-SFT.** Evaluation uses the held-out BIRD Setting 1 (outcome-changing edits) instances (N=30). **OC-edit** denotes ASK supervision from outcome-changing edits; **OP-edit** denotes NOASK supervision from Setting 3 outcome-preserving edits; and **Paired** denotes the auxiliary paired-original NOASK pool. "Balanced" denotes taxonomy-guided balancing of ASK examples. Train A/N reports the effective number of ASK/NOASK training examples after sampling. The bold row is the final LAMP-SFT model reported in the main paper. The three no-NOASK rows (NOASK source "None") retain the corresponding ablation family's validation protocol.

For the learning study, 100 of the 500 BIRD mini-dev questions are reserved for held-out evaluation, and supervision is derived only from the remaining questions. Applying the controlled-edit construction of Section 5 to the held-out questions yields 30 Setting 1 (outcome-changing) and 50 Setting 3 (outcome-preserving) instances, reported in Table 9 and Appendix D, respectively. Training and evaluation are disjoint at the base-question level, including all edited variants of each question.

G.2 Training and decoding

We fine-tune Qwen2.5-Coder-7B-Instruct with 4-bit QLoRA. LoRA rank is 16 with $\alpha = 32$ and dropout 0.05, applied to the attention and MLP projection layers. The model is optimized with paged AdamW at learning rate 2×10^{-4}, effective batch size 16, cosine decay, and a maximum of 12 epochs with early stopping. Training uses one NVIDIA A100 80GB GPU.

Only assistant tokens are supervised. The target begins with the decision token (ASK or NOASK); ASK targets then include one or more anchor spans, each with up to four candidate readings. We use grouped cross-entropy so that the decision term is not diluted by the longer span-and-reading sequence. At inference time, the first-token probabilities define $P(\text{ASK})$, and the ASK threshold is selected on a validation split using F_2. Conditional on ASK, the spans and readings are decoded greedily. The validation split is drawn from the training pool only: a BIRD question is assigned to validation if the SHA-256 hash of its question identifier is divisible by 5, so all edited variants of a question fall on the same side of the split. For the final configuration, it contains 140 examples (106 ASK, 34 NOASK) and is used to select the checkpoint and the ASK threshold. It is disjoint from the held-out evaluation instances (30 Setting 1 and 50 Setting 3), which are never used in training or validation.

G.3 Supervision ablations

Table 9 compares the source and composition of the supervision used to train the clarification policy. With the same outcome-preserving NOASK source, LAMP-derived ASK supervision yields higher Target Recall than supervision from constructed outcome-changing edits: 0.70 versus 0.43 without ASK balancing and 0.73 versus 0.33 with balancing. The final balanced ASK:NOASK configuration reaches 0.67 Target Recall, 0.46 Target Precision, 0.63 execution accuracy, and 1.53 clarification questions per instance.

Method	Target Rec. ↑	Target Prec. ↑	Exec Acc. ↑	n_q ↓
LAMP-OneShot	0.53	0.24	0.47	2.27
LAMP-Enumerate	0.43	0.37	0.43	1.13
LAMP-Verify	0.33	0.32	0.33	0.87
LAMP-SFT	0.67	0.46	0.63	1.53

Table 10: **Backbone sensitivity on held-out Setting 1 (N=30).** All rows use the same downstream execution stack. The LAMP-SFT row is the final selected checkpoint from the supervision ablation.

H LIMITATIONS

Assumptions are tested one at a time. Our formulation treats unresolved assumptions as independently resolvable, assuming that every combination of readings is jointly valid (Section 3), and both LAMP-Verify and the evaluation construction change one assumption at a time. In practice, the readings of different assumptions can constrain one another, and an assumption may affect the outcome only in combination with another. Single-detail edits and per-assumption verification do not test these interactions.

Candidate coverage depends on backbone capacity. We run the prompted LAMP stages with Qwen2.5-Coder-7B-Instruct as π_{clar} while keeping the downstream stack fixed. With this smaller model, the later stages become more selective but lose recall, revealing a bottleneck in upstream candidate discovery.

With Qwen2.5-Coder-7B-Instruct as π_{clar} (Table 10), Target Recall falls from 0.53 for LAMP-OneShot to 0.43 for LAMP-Enumerate and 0.33 for LAMP-Verify as the pipeline becomes more selective. Verification can filter discovered candidates but cannot recover a target assumption that was never proposed upstream. The fine-tuned LAMP-SFT model instead reaches 0.67 recall at 0.46 precision and 1.53 questions on the same instances. This exposes a concrete limitation of the prompted multi-stage pipeline: its filtering stages depend on sufficient candidate coverage from π_{clar}, and supervised policy transfer offers a complementary route when that coverage is weak.

Outcome preservation is contextual. A Setting 3 case shows that an omission does not change the outcome for the tested agent, context, and execution protocol; it does not show that the corresponding assumption is non-load-bearing in general (Appendix C.3).

Simulated users and execution cost. User responses are simulated, so our results do not measure how people answer clarification questions. LAMP-Verify also requires an executable environment and a task-level equivalence test, and its repeated executions increase cost (0.26 USD per instance on Setting 1, compared with 0.03 for Ambig-SWE; Table 1).