Know Before Fix: QA-Driven Repository Knowledge Acquisition for Software Issue Resolution

Haotian Lin[1], Silin Chen[1], Xiaodong Gu[1*], Yuling Shi[1],
Chengxi Pan[2], Jiaqi Ge[3], Mengfan Li[1],
Jianghong Huang[1], MENGCHIEH CHUANG[1], Beijun Shen[1], and Haibing Guan[1]
[1]Shanghai Jiao Tong University, Shanghai, China
[2]University of Pittsburgh, Pittsburgh, PA, USA
[3]Guangdong Technion–Israel Institute of Technology, Shantou, China
{hyperlynnx, xiaodong.gu, yuling.shi, lmf2951510526, jh.huang, Zhuangmengjie, bjshen, hbguan}@sjtu.edu.cn,
csl2457029646@163.com, chp252@pitt.edu, ge24876@gtiit.edu.cn

***Abstract*—LLM-based coding agents have significantly advanced automated software issue resolution, yet they remain highly prone to factual errors caused by insufficient repository understanding. Recent methods attempt to mitigate this limitation through pre-repair repository exploration; however, their fix-driven strategies explore repositories without identifying the agent's knowledge gaps, often yielding imprecise context that fails to bridge the underlying understanding deficit. In this paper, we propose ACQUIRE, a QA-driven framework for software issue resolution. Mirroring how experienced developers first comprehend unfamiliar code before attempting a fix, ACQUIRE explicitly acquires repository knowledge prior to repair. The framework decouples knowledge acquisition from patch generation through two stages: in the first stage, a *Questioner* and an *Answerer* collaborate to acquire structured repository knowledge, where the *Questioner* poses targeted questions and the *Answerer* produces evidence-grounded answers through autonomous exploration; in the second stage, the *Resolver* leverages the resulting QA knowledge to generate informed patches. By transforming implicit knowledge gaps into explicit, factually reliable understanding, ACQUIRE accelerates knowledge-intensive repair stages and enables more accurate resolution. Experiments on SWE-bench Verified demonstrate that ACQUIRE consistently outperforms representative pre-repair methods, raising Pass@1 by up to 4.4 percentage points with modest additional cost and time.**

***Index Terms*—software issue resolution, repository knowledge acquisition, coding agents, large language models**

I. Introduction

Coding agents empowered by large language models (LLMs) have driven significant advances in automated software issue resolution [1]–[3]. Powered by techniques such as chain-of-thought reasoning, multi-step planning, and tool-augmented interaction, these agents can navigate complex codebases, identify relevant code locations, and generate candidate patches with remarkable fluency [4]–[6].

Despite these rapid advancements, a critical class of failures remains largely unresolved, even for frontier-scale models: *factual errors* stemming from *insufficient repository understanding*. These failures are not due to limitations in the model's reasoning capacity; rather, they arise because the issue description alone does not supply the repository-internal knowledge, such as cross-module dependencies, implicit API contracts, and data-flow details, required to formulate a correct fix. Consequently, agents often fall back on shallow, keyword-based localization, fail to trace fault origins across module boundaries, and inadvertently violate implicit API contracts [6]–[9]. Furthermore, these knowledge-deficient attempts are disproportionately costly, consuming over four times the token cost and nearly twice the execution steps compared to successful resolutions [8], [10].

To bridge this gap, recent methods have attempted to explore the repository prior to issue resolution [1], [3], [11], [12]. For instance, LingmaAgent links issue keywords to code entities and constructs structural summaries to provide broader codebase context [12]. While effective for straightforward bugs, these techniques are fundamentally driven by issue keywords rather than by deep understanding of the repository, and the resulting context is frequently incomplete or imprecise, ultimately compounding the agent's misunderstanding of the codebase [7], [9].

We propose **ACQUIRE** (**A**gent **C**ollaboration for **Q**uestion-Answer-driven **I**ssue **RE**solution), a framework that explicitly decouples repository knowledge acquisition from patch generation. Mirroring how experienced developers first build understanding of an unfamiliar codebase before attempting a fix, ACQUIRE explicitly identifies what repository knowledge the agent lacks and *proactively* acquires it before repair begins. The framework orchestrates three specialized agents across two stages. In the knowledge acquisition stage, a Questioner and an Answerer collaborate to acquire structured repository knowledge: the Questioner decomposes the issue into targeted questions across complementary knowledge dimensions, and the Answerer resolves each ques-

* Xiaodong Gu is the corresponding author.

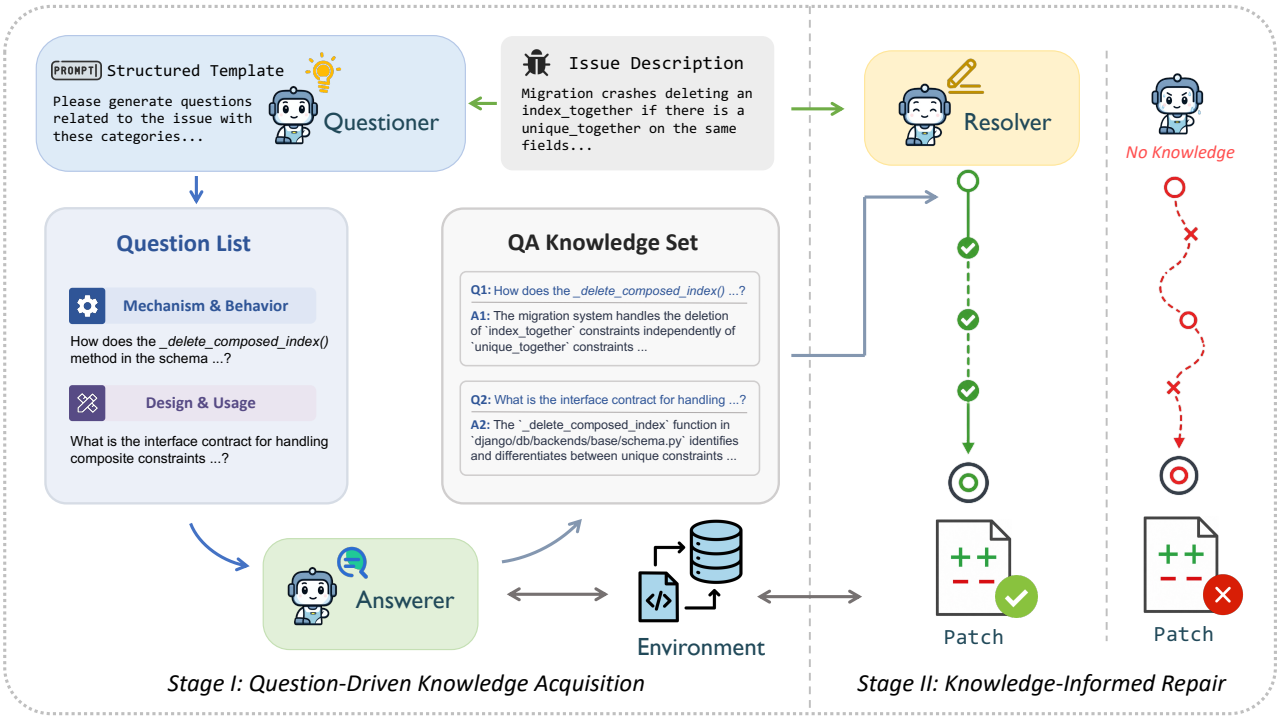

Fig. 1. Overview of ACQUIRE.

tion through autonomous repository exploration, producing evidence-grounded QA pairs. In the repair stage, the Resolver leverages this structured QA knowledge to generate informed patches. This design transforms implicit knowledge gaps into explicit, structured understanding before editing begins.

Experiments on SWE-bench Verified [13], a curated benchmark of 500 real-world GitHub issues, demonstrate that ACQUIRE raises Pass@1 by up to 4.4 percentage points over the base repair agent, consistently outperforming all representative pre-repair methods across two backbone LLMs at modest additional cost and time.

Our contributions are threefold:

- We propose ACQUIRE, a QA-driven framework for software issue resolution. To our knowledge, this is the first work to integrate repository-level QA into issue resolution.
- ACQUIRE decouples knowledge acquisition from patch generation into two explicit stages, and introduces a category-guided question taxonomy that enables targeted acquisition of the repository knowledge needed for repair.
- We provide an in-depth analysis showing that the acquired knowledge is factually reliable, reduces repair trajectory length, and steers the agent toward causally relevant code regions on previously failed instances, revealing how pre-repair knowledge improves resolution.

II. Motivation

Recent empirical studies [6]–[9] have consistently identified **insufficient repository knowledge** as a dominant cause of SE agent failure, manifesting as shallow keyword-based localization, failure to trace faults across module boundaries, violation of implicit API contracts, and missing knowledge of external protocols [6], [7], [9]. Existing pre-repair methods attempt to mitigate this gap but remain fundamentally *fix-oriented*: whether producing ranked suspicious locations [3], [11] or enriching context with structural summaries [1], [12], they rely on the agent's own ability to convert shallow pointers into correct patches, and neither family explicitly identifies *what repository knowledge the agent still lacks* before repair begins [7], [9]. This contrasts with how experienced developers approach unfamiliar bugs: rather than immediately searching for a fix, they first ask diagnostic questions about the code, such as how a component behaves, what contracts an API enforces, or where relevant modules reside, building the necessary understanding before committing to any repair strategy [14], [15].

Inspired by recent work on repository-level question answering [16], we pose a different question: if an agent could decompose an issue into targeted, *easy-to-answer* repository questions and obtain accurate answers *before* any repair attempt, could such high-quality knowledge more effectively unlock its repair potential? To test this hypothesis, we design an *oracle QA* experiment, adopting the oracle probing setup of [17]. Concretely, we construct an oracle Questioner that takes the issue description together with the golden test patch as input and generates a single question most relevant to resolving the issue. An oracle Answerer then receives only the files involved in the golden patch and uses them as grounding context to answer the question. We inject the resulting oracle

QA pair into Mini-SWE-Agent [18] and evaluate on the 116 instances from SWE-bench Lite [19] that Mini-SWE-Agent fails to resolve under DeepSeek-V3.2 [20]. With oracle QA injection, 26 of these 116 previously failed instances are successfully resolved, indicating that targeted pre-repair knowledge can bridge the gap between the agent's reasoning capacity and the repository understanding required for correct fixes. Note that this oracle setting is intentionally constrained, as only one question is asked and the oracle Answerer can only access the gold files rather than the full repository, so the 26/116 recovery rate reflects this restricted setup rather than an inherent ceiling of the QA paradigm.

While the oracle experiment confirms the promise of question-driven knowledge injection, it relies on privileged information, namely the golden test patch and the golden file set, which is unavailable in real-world settings. To bridge this gap toward a fully automated pipeline, we address each oracle dependency in turn. For the Questioner, five computer science graduate students manually analyzed the 116 oracle questions from the above experiment and, cross-referencing with the failure taxonomies established in the aforementioned studies [6]–[9], identified four recurring question patterns that cover distinct dimensions of the repository knowledge needed for repair: (1) *Mechanism & Behavior* (70.7%), targeting how internal logic flows and data transformations actually work; (2) *Design & Usage* (18.1%), targeting what API contracts and design conventions govern valid edits; (3) *Locating & Structure* (7.8%), targeting where relevant code resides and how modules are organized; and (4) *Ecosystem & Standards* (3.4%), targeting what external library behaviors or protocol constraints apply. Together, they serve as the structured question template, enabling the Questioner to generate targeted questions without any oracle information. For the Answerer, we replace the privileged file set with an autonomous agent that explores the repository, discovering relevant files and gathering context on its own. The detailed design of both components is presented in Section III.

III. Methodology

A. Overview

We propose ACQUIRE, a two-stage framework that mirrors how experienced developers approach an unfamiliar codebase: first acquiring the necessary repository knowledge, then performing an informed repair. As illustrated in Figure 1, given an issue description $\mathcal{I}$ and an execution environment $\mathcal{E}$, the Questioner first generates N targeted questions, and N independently instantiated Answerer instances explore the repository in parallel to answer them, producing a knowledge set $\mathcal{K} = \{(q_1, a_1), \ldots, (q_N, a_N)\}$. The Resolver then leverages $\mathcal{I}$ and $\mathcal{K}$ to generate a candidate patch. The following subsections detail each stage.

B. Stage 1: Question-Driven Knowledge Acquisition

Stage 1 acquires structured repository knowledge through N question–answer interactions before any repair action begins.

1) Question Generation: To ensure the questions cover knowledge dimensions most relevant to repair, the Questioner is guided by a structured prompt template derived from the knowledge-deficit patterns identified in Section II. The template encodes the four category descriptions with concrete examples, a JSON output schema, and constraints enforcing non-redundancy and self-containedness (the full template is provided in the supplementary material). The four categories are defined as follows, where the first three target distinct dimensions of *repository-internal* knowledge, while the fourth addresses *external* knowledge that the repository depends on but does not contain:

- **Mechanism & Behavior.** Targets functional logic flows, state management, and data processing details, asking *how a functionality actually works*, e.g., "How does the authentication system validate user credentials in this repository?" Such questions are critical for tracing the root cause of behavioral bugs.
- **Design & Usage.** Targets API definitions, class hierarchies, error specifications, and design conventions, asking *how interfaces are designed and what contracts they follow*, e.g., "What is the interface contract for the database connection class?" These questions help the downstream agent avoid violating existing design constraints.
- **Locating & Structure.** Targets the codebase layout, module organization, and dependency relationships, asking *where things are and how they relate*, e.g., "Where is the login functionality implemented?" These questions directly reduce the search space for repair.
- **Ecosystem & Standards.** Targets external knowledge such as third-party library behaviors, protocol specifications, and language-level semantics, asking *what external knowledge is needed*, e.g., "What does the HTTP/1.1 specification require for chunked transfer encoding?" These questions help the repair agent reason correctly about constraints beyond the codebase.

The four categories serve as a guiding taxonomy rather than a rigid per-question constraint. The Questioner autonomously selects the most appropriate category for each question based on the issue context; no external scheduling or round-robin policy is imposed. Consequently, a single category may recur across questions when the most critical knowledge gaps fall under it.

2) Evidence-Grounded Answering: For each question, the Answerer produces a grounded answer through autonomous repository exploration. The Answerer shares the same scaffolded shell interaction environment as Mini-SWE-Agent but operates in read-only mode, ensuring that no repository state is modified during knowledge acquisition. Its prompt specifies three behavioral requirements: (1) answers must reference concrete repository artifacts such as file paths, function names, and relevant code behaviors, rather than relying on speculation or parametric knowledge; (2) the agent should actively submit its answer once sufficient evidence has been gathered; and

(3) if sufficient evidence cannot be found, the agent should explicitly acknowledge the gap rather than fabricate unsupported claims. In practice, all instances completed within the allocated budget without forced truncation. The factual reliability of the acquired answers is evaluated in Section V-B1, and a characterization of the generated QA is provided in the supplementary material.

3) Parallel Knowledge Acquisition: Given the issue description $\mathcal{I}$, the Questioner generates N questions $\{q_1, \ldots, q_N\}$. All N questions are then dispatched to N independently instantiated Answerer instances that explore $\mathcal{E}$ in parallel, each producing a grounded answer a_i for its assigned question q_i.

Each Answerer instance receives only $\mathcal{I}$ and its assigned question q_i as input, without access to other questions or exploration traces. This isolation serves two purposes: it prevents cross-instance context from biasing exploration toward previously visited but potentially irrelevant code regions, keeping each answer focused and independently grounded; and it enables fully parallel execution, reducing the wall-clock latency of the knowledge acquisition stage to that of the single slowest instance rather than the sum of all instances. After all N instances complete, the resulting pairs are assembled into the knowledge set $\mathcal{K} = \{(q_1, a_1), \ldots, (q_N, a_N)\}$ and passed to Stage 2.

C. Stage 2: Knowledge-Informed Repair

In Stage 2, the Resolver performs issue resolution guided by the repository knowledge acquired in Stage 1. The Resolver receives the issue description $\mathcal{I}$ together with the knowledge set $\mathcal{K}$, and operates within $\mathcal{E}$ through an iterative loop of code navigation, editing, and test execution until a candidate patch is produced.

The N QA pairs are serialized into a structured text block and prepended to the Resolver's messages before the repair instruction, so that repository understanding is fully established before the first repair action and remains stable throughout the trajectory. We adopt static pre-injection rather than dynamic interleaving (i.e., streaming QA content into the trajectory on demand) for two reasons: (1) it preserves the modularity of the two-stage design by avoiding coupling between knowledge acquisition and the repair loop; and (2) it ensures the complete knowledge context is available from the first repair step, preventing early-stage decisions from being made under partial information. Importantly, the injected QA serves as supplementary rather than prescriptive context, informing the Resolver's strategies without overriding its ability to independently verify and adjust based on direct repository observations. This minimal-intervention design ensures that any performance difference is attributable to the acquired knowledge rather than to changes in the repair mechanism itself. Trajectory analysis shows that the agent consults QA knowledge primarily in knowledge-intensive phases, using it to narrow file search during localization and to inform edit decisions during fixing, while relying more on executable feedback in reproduction and verification phases (Section V-B2).

IV. Experimental Setup

A. Research Questions

We evaluate ACQUIRE by addressing the following research questions:

RQ1 (Effectiveness): How effective is ACQUIRE for issue resolution?

RQ2 (Knowledge Quality & Influence): How effectively does the generated QA knowledge support repair behavior?

RQ3 (Ablation): How much do the question decomposition and the category-guided question generation each contribute to repair? **RQ4 (Number of QA):** How does the number of QA pairs affect repair performance and cost?

B. Datasets and Models

We evaluated our method on SWE-bench Verified [13], a high-quality subset of SWE-bench [19], containing 500 real GitHub issues focused on fixing functional bugs in a controlled, isolated environment. For each instance, the model only receives a problem description in natural language and the corresponding code repository. The correctness of the generated patches is evaluated by running unit tests written by the developers, thus providing a consistent and rigorous evaluation of the performance of automated bug fixing.

To assess the generality of ACQUIRE across model families, we adopt two backbone LLMs that have been widely used to evaluate SE agents [21]–[24], representing distinct development paradigms:

- **DeepSeek-V3.2** [20]: an open-source model built on a Mixture-of-Experts (MoE) architecture with 671B total parameters and approximately 37B activated per token. It incorporates sparse attention mechanisms and large-scale reinforcement learning post-training, achieving frontier-level performance in code understanding and generation tasks.
- **GPT-5-mini** [25]: a proprietary, efficiency-oriented model from OpenAI, optimized for high throughput and low-latency deployment while maintaining strong general-purpose capabilities.

By pairing an open-source reasoning-specialized model with a closed-source efficiency-oriented model, our evaluation covers two representative points in the current LLM landscape.

C. Evaluation Metrics

We evaluate all methods using two primary metrics that jointly capture resolution effectiveness and economic cost:

- **Pass@1**: the proportion of instances successfully resolved in a single attempt, serving as the primary effectiveness metric.
- **Average Cost**: the average monetary cost per instance over the entire pipeline, which includes both the pre-repair exploration stage and the repair stage.
- **Average Time**: the average end-to-end wall-clock time per instance, measured from the start of the pre-repair stage to the completion of patch generation.

D. Baseline Methods

We compare ACQUIRE with representative pre-repair exploration methods, alongside Mini-SWE-Agent which serves as the shared base repair agent.

- **Mini-SWE-Agent** [18]: a minimal agentic repair system derived from SWE-agent [4] that iteratively proposes shell actions, observes execution results, and updates its strategy until a patch is produced. It has been widely adopted as a standardized repair backbone in recent evaluations [2], [9], [10], [21]; in our study, it serves both as a standalone baseline and as the shared base repair agent for all compared methods. Its minimal design makes it straightforward to attribute performance differences to the methods themselves.
- **LocAgent** [3]: constructs a heterogeneous repository graph encoding import, call, and inheritance relations, then performs LLM-guided multi-hop traversal to produce a ranked list of suspicious fault locations.
- **CoSIL** [11]: incrementally expands a local call graph during search, iteratively refining the search frontier and pruning context to yield compact, high-relevance code fragments as pre-repair evidence.
- **LingmaAgent** [12]: builds a repository-level knowledge graph with a summarized global view, then uses Monte Carlo Tree Search (MCTS) to guide exploration, producing fine-grained fault localization signals and a repair-oriented repository summary.
- **SWE-Debate** [1]: introduces multi-agent debate into the repair pipeline, where multiple LLM agents independently propose repair hypotheses, iteratively challenge each other's reasoning, and converge on a consensus repair plan that guides downstream patch generation.

E. Implementation Details

1) Baseline Method Configurations: For **Mini-SWE-Agent**, we follow the official SWE-bench setup. The agent runs in an isolated Docker container, a per-step command timeout of 360 seconds, a maximum trajectory length of 250 steps, a per-instance cost cap of $3.00, and temperature 0.0. In our implementation, all baselines and ACQUIRE adopt Mini-SWE-Agent as the base repair agent and share the same repair-stage settings. For all other baseline methods, we follow the default settings reported in their original papers.

2) ACQUIRE Configurations: The knowledge acquisition stage generates $N{=}2$ questions per issue, with Questioner using temperature 0.7 to encourage diverse and complementary coverage across the four question categories. Answerer uses the same scaffolded interaction environment as Mini-SWE-Agent with read-only execution, a per-step timeout of 120 seconds, a maximum trajectory length of 150 steps, a per-instance cost cap of $2.00, and temperature 0.0. Resolver inherits the same scaffold, execution environment, and settings as the baseline Mini-SWE-Agent. Due to space limitations, the prompt templates are provided in the supplementary material.

TABLE I
MAIN RESULTS ON SWE-BENCH VERIFIED UNDER GPT-5-MINI AND DEEPSEEK-V3.2.

Method	Model	Pass@1 (%)	Avg Cost ($)	Avg Time (s)
Mini-SWE-Agent	GPT-5-mini	58.4	0.024	187
	DeepSeek-V3.2	66.4	0.055	815
LocAgent	GPT-5-mini	57.8 (↓0.6)	0.048	437
	DeepSeek-V3.2	<u>68.8</u> (↑2.4)	0.060	1046
CoSIL	GPT-5-mini	55.2 (↓3.2)	0.035	224
	DeepSeek-V3.2	68.7 (↑2.3)	0.059	750
LingmaAgent	GPT-5-mini	<u>60.0</u> (↑1.6)	0.309	823
	DeepSeek-V3.2	67.4 (↑1.0)	0.155	1421
SWE-Debate	GPT-5-mini	59.8 (↑1.4)	0.738	1552
	DeepSeek-V3.2	68.2 (↑1.8)	0.382	2517
ACQUIRE (Ours)	GPT-5-mini	**62.2** (↑3.8)	0.054	302
	DeepSeek-V3.2	**70.8** (↑4.4)	0.073	1042

V. RESULTS

A. RQ1: Effectiveness of ACQUIRE

Table I reports the main results. Arrows indicate improvement or degradation relative to Mini-SWE-Agent, and underlines mark the best-performing baseline for each backbone model.

Among the baselines, LocAgent and CoSIL achieve moderate gains under DeepSeek-V3.2 but degrade under GPT-5-mini. This cross-model inconsistency highlights a limitation of localization-oriented approaches: they provide shallow code pointers (e.g., suspicious file lists or code fragments) and rely on the backbone model to infer *why* these locations are relevant. When the model lacks this reasoning capacity, such signals can mislead the repair process. In contrast, LingmaAgent and SWE-Debate combine static analysis with LLM-driven repository traversal. LingmaAgent employs MCTS-based planning to produce repair-oriented summaries, while SWE-Debate leverages multi-agent debate to converge on a consensus repair plan. Both yield consistent improvements across models, confirming the value of richer pre-repair context. However, their extensive traversal substantially increases cost and time, limiting practical deployment.

ACQUIRE outperforms all baselines by a clear margin. First, it consistently achieves the largest Pass@1 gains across both backbone models (+3.8 on GPT-5-mini, +4.4 on DeepSeek-V3.2), demonstrating strong cross-model generalization; in contrast, localization-oriented baselines even degrade under GPT-5-mini. Second, unlike localization baselines that supply shallow code pointers, the QA-driven paradigm captures richer knowledge dimensions such as behavioral context, design constraints, and structural relationships, as further characterized in the supplementary material. While LingmaAgent and SWE-Debate also produce richer context, they do so at 1.4–5$\times$ longer end-to-end time and 2–14$\times$ higher cost. Third, ACQUIRE maintains competitive efficiency, with end-to-end time and cost comparable to the lightweight localization baselines. This favorable accuracy–efficiency trade-off stems from the QA-driven paradigm, which elicits targeted knowledge without exhaustive repository traversal.

Finding 1: ACQUIRE achieves the highest Pass@1 across both GPT-5-mini (+3.8) and DeepSeek-V3.2 (+4.4), with modest cost and time overhead. These results demonstrate that targeted QA-driven knowledge acquisition is more effective, more generalizable, and more efficient than existing pre-repair strategies.

B. RQ2: Quality and Influence of QA Knowledge

Having established that ACQUIRE consistently improves resolution rates, this section examines the reliability of the generated QA knowledge and its influence on downstream repair behavior. To control for model variation, all analyses in this section use DeepSeek-V3.2.

1) QA Factual Reliability: As shown in Table I, comparing per-instance outcomes between Mini-SWE-Agent and ACQUIRE yields 44 Fail→Pass recoveries and 22 Pass→Fail regressions, producing a net gain of 22 resolved instances. To systematically assess the factual reliability of the generated QA, we conduct a human audit on 232 QA pairs drawn from all four transition cells: all pairs from the 44 Fail→Pass and 22 Pass→Fail cases, plus 25 randomly sampled instances each from the Fail→Fail and Pass→Pass cells. Four computer-science master's students, each with four years of development experience, served as reviewers. Each QA pair was independently reviewed by two reviewers, with disagreements resolved through discussion.

Of the 232 audited QA pairs, 230 (99.1%) are labeled *Supported*: 98 (42.2%) are fully accurate with all claims grounded in repository evidence, and 132 (56.9%) contain minor deviations where core claims are correct but local details such as line numbers or function ownership are imprecise, none of which affect the overall conclusion. Only 2 pairs (0.9%) contain ungrounded claims where the central mechanism assertion is contradicted by the repository code. A detailed breakdown of deviation types and full descriptions of the two ungrounded cases are provided in the supplementary material.

This high factual reliability stems from ACQUIRE's QA-driven design: by decomposing a complex issue into targeted, narrowly scoped questions, each Answerer instance faces a substantially simpler retrieval and reasoning task than answering the full issue at once, reducing the opportunity for hallucination and enabling evidence-grounded answers. The ablation in Section V-C1 further confirms this decomposition benefit.

2) Influence on Repair Behavior: To investigate how QA reshapes downstream repair, we compare Mini-SWE-Agent without any pre-repair knowledge (**w/o QA Injection**) against the same configuration augmented with QA pairs produced by ACQUIRE (**w/ QA Injection**). QA injection reduces the mean number of agent rounds by 7.1% on the full 500 instances and by 17.1% on the 44 Fail→Pass instances, confirming that front-loaded repository knowledge bypasses expensive trial-and-error loops [8], [10]. The full round-distribution plots and a per-transition API-call shift analysis are provided in the supplementary material.

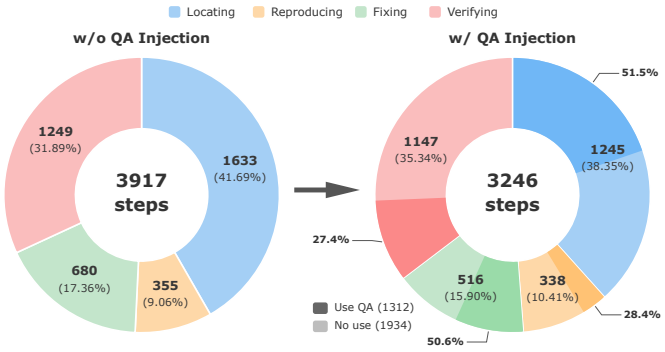

Fig. 2. Trajectory stage composition on 44 *Fail→Pass* instances w/o vs. w/ QA Injection. Darker shading in the right ring marks QA-related steps.

Since the acceleration is most pronounced on Fail→Pass instances, we focus the remaining trajectory analysis on these 44 cases to study *where* within each trajectory QA knowledge takes effect. We divide each repair trace into four canonical stages [5], [7], [8] and annotate whether each step is QA-related, as shown in Figure 2. A step is labeled QA-related if the agent explicitly references QA content, or if its action closely follows QA-provided evidence (e.g., opening a file identified in a QA answer or verifying a QA-described behavior). The annotation follows the same protocol described in Section V-B. The total number of repair steps drops from 3,917 to 3,246, but the reduction is concentrated in the knowledge-intensive phases: Locating contracts by 23.8% and Fixing by 24.1%, while Reproducing and Verifying *grow* in proportion (+1.4 pp and +3.5 pp). QA knowledge accelerates locating relevant code and composing the fix, freeing the agent to dedicate a larger share of effort to reproduction and verification. This shift parallels the behavior of expert human debuggers, who invest upfront effort in program comprehension and consequently require less trial-and-error exploration [14], [15].

The step-level annotations confirm this pattern: 40.4% of all steps are QA-related, with usage concentrated in Locating (51.5%) and Fixing (50.6%) compared with Reproducing (28.4%) and Verifying (27.4%). The two most accelerated stages are precisely the ones where the agent draws most heavily on QA, providing direct evidence that the speedup is driven by the injected knowledge.

3) Pass-to-Fail Regression Analysis: Although ACQUIRE produces a net gain of 22 resolved instances and the preceding analysis demonstrates that QA knowledge substantially reshapes repair behavior, the 22 Pass→Fail regressions warrant closer examination. We manually inspect all 22 regression trajectories and classify each case by whether the injected QA played a misleading role in the repair failure. A case is labeled *misleading* when the dominant downstream effect of the injected QA on the Resolver's behavior leads it away from the correct fix.

Under this criterion, only 5 of the 22 regressions are misleading: the QA emphasizes a related but non-gold loca-

TABLE II
ABLATION RESULTS ON DOWNSTREAM REPAIR PERFORMANCE ON SWE-BENCH VERIFIED UNDER DEEPSEEK-V3.2.

Method	Pass@1 (%)	Δ
ACQUIRE	**70.8**	–
ACQUIRE-Proposal	66.0	↓4.8
ACQUIRE-FreeQ	67.0	↓3.8

tion or mechanism, and the Resolver repeatedly follows this framing during search and editing, eventually deviating from the correct fix. Notably, cross-referencing with the factual audit reveals that among the 10 QA pairs in these 5 cases, only 1 contains an ungrounded claim; the remaining 9 are factually Supported. This indicates that misleading regressions arise primarily from QA providing a plausible but incorrect repair framing, rather than from factual errors in the answers themselves. The remaining 17 regressions are non-misleading: the QA is constructive or neutral, and the failures stem from Resolver-side behavior such as overgeneralizing a correct clue, implementing only part of the required fix, or introducing unnecessary modifications. Detailed behavioral breakdowns for both groups are provided in the supplementary material.

This analysis reveals that the regression bottleneck lies in how the Resolver utilizes otherwise reliable QA, not in QA quality itself, suggesting that improving the Resolver's capacity to critically evaluate and selectively apply injected knowledge is a promising direction for reducing regressions.

Finding 2: The generated QA knowledge is highly reliable, with the vast majority of pairs factually supported by repository evidence. This reliable knowledge accelerates repair by concentrating speedup in the knowledge-intensive Locating and Fixing stages, while the few regressions stem primarily from Resolver-side misuse rather than QA factual errors.

C. RQ3: Ablation on Key Design Choices

To validate the contribution of each key design in ACQUIRE, we construct two ablation variants and evaluate them on SWE-bench Verified under DeepSeek-V3.2. Each variant removes one core component while keeping the rest of the pipeline unchanged:

- **ACQUIRE-Proposal** replaces the Questioner–Answerer QA pipeline with a single proposal-generation step. For fairness, the proposal agent shares the same instruction scaffold as Answerer; only the task differs: it reads the issue and directly produces a solution proposal containing root-cause analysis and suggested fix strategy, instead of answering a targeted repository question. This proposal substitutes for the structured QA pairs injected into Resolver.
- **ACQUIRE-FreeQ** retains the full QA pipeline but removes the category-driven question template from Questioner. Rather than following the four prescribed question categories, Questioner freely generates N (N=2 by default) questions without any categorical constraint.

Table II reports the results. The following two subsections examine each variant in detail.

1) QA Decomposition vs. Proposal: As shown in Table II, ACQUIRE-Proposal causes the largest performance drop among all ablation variants, reducing Pass@1 by 4.8 percentage points to 66.0%. Notably, this score even falls below that of Mini-SWE-Agent, i.e., the repair agent without any pre-repair information (66.4%, Table I), meaning the injected proposal not only fails to help but actively *degrades* downstream repair. The root cause is that a single-pass proposal must simultaneously identify root causes, reason about mechanisms, and suggest a fix strategy, producing an analysis that tends to be superficial and entangled when the issue is complex. By contrast, ACQUIRE decomposes the issue into narrowly scoped questions, each isolating one specific aspect and allowing the corresponding answer to be reliably grounded through focused exploration. This structured decomposition is a critical enabler of ACQUIRE's performance.

2) Category-Guided vs. Free Question Generation: Removing the category-driven template (ACQUIRE-FreeQ) also leads to a notable decline, with Pass@1 dropping by 3.8 percentage points to 67.0%. To understand *why*, we directly analyze question quality using an LLM-as-a-judge protocol [26]–[28] with DeepSeek-V3.2, evaluating five question-level dimensions and one set-level dimension (*coverage*). Each judgment is scored on a 1–10 scale and averaged over 10 repetitions. Full scoring criteria and evaluation prompts are provided in the supplementary material.

TABLE III
QUESTION-QUALITY COMPARISON BETWEEN ACQUIRE-FREEQ AND ACQUIRE ON SWE-BENCH VERIFIED.

Metric	ACQUIRE-FreeQ	ACQUIRE	Δ
Relevance	8.22	**8.49**	↑0.27
Answerability	**8.65**	8.64	↓0.01
Diagnostic Utility	7.58	**7.96**	↑0.38
Reasoning Depth	6.90	**7.28**	↑0.38
Clarity	8.42	**8.50**	↑0.08
Coverage	5.97	**6.74**	↑0.78

As shown in Table III, the category-guided questions achieve the most prominent advantages in *diagnostic utility* (+0.38), *reasoning depth* (+0.38), and *coverage* (+0.78), while *answerability* is virtually identical (−0.01). The largest gain in *coverage* confirms that the category-driven template effectively prevents questions from clustering around a single diagnostic angle. A complementary pairwise voting analysis [27] further corroborates this: ACQUIRE is preferred in 294 of 500 comparisons versus 111 for ACQUIRE-FreeQ (95 ties), yielding a non-loss rate of 77.8%.

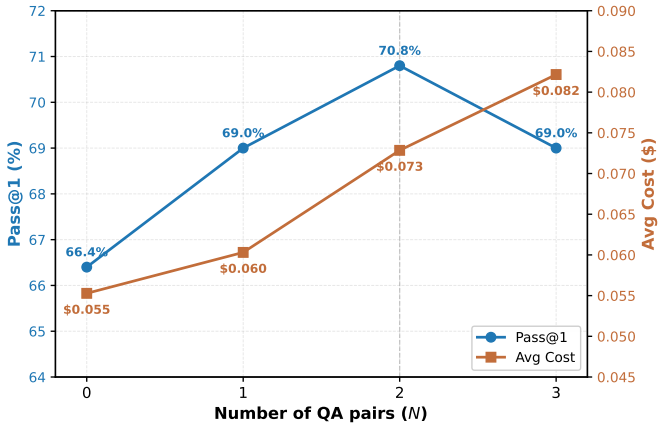

Fig. 3. Pass@1 and average cost per instance under varying the number of QA pairs.

Finding 3: Replacing question decomposition with a single-pass proposal drops Pass@1 by 4.8 pp, and removing the category-driven template drops it by 3.8 pp with notably lower question quality, confirming both designs as essential to ACQUIRE.

D. RQ4: Sensitivity to the Number of QA Pairs

We investigate how N, the number of QA pairs generated per issue, affects repair performance and cost.

Figure 3 reports Pass@1 and average cost as functions of N. When $N{=}0$, no QA is injected and the pipeline degrades to the Resolver-only baseline (i.e., Mini-SWE-Agent), which achieves 66.4% Pass@1 at \$0.055 per instance. Introducing even a single QA pair ($N{=}1$) raises Pass@1 by 2.6 percentage points (from 66.4% to 69.0%) with only \$0.005 additional cost, demonstrating that the QA-driven paradigm within ACQUIRE delivers meaningful improvement under only one pair of QA knowledge. Across all $N{\geq}1$ settings, ACQUIRE consistently outperforms every baseline reported in Table I, confirming its robustness to the number of QA pairs and highlighting the superiority of semantically rich repository knowledge over alternative forms of repository information such as localization pointers or structural summaries.

Pass@1 peaks at $N{=}2$ (70.8%) and then drops back to 69.0% at $N{=}3$, while average cost continues to rise. We attribute this non-monotonic pattern to two factors. First, two questions from complementary categories cover distinct knowledge gaps more effectively than a single question. Second, a third pair tends to overlap with already-covered knowledge, and the longer injected context may dilute the repair agent's attention rather than strengthen it, yielding diminishing returns [29]–[31]. Based on this cost–performance profile, we adopt $N{=}2$ as the default setting for all other experiments.

Finding 4: ACQUIRE consistently outperforms all baselines across all tested $N{\geq}1$ settings, with performance peaking at $N{=}2$, which offers the best trade-off between effectiveness and cost.

E. Case Study

We illustrate the effectiveness of ACQUIRE through a representative example from SWE-bench Verified under DeepSeek-V3.2: `sphinx-doc__sphinx-9230` (Figure 4). The issue reports that Sphinx incorrectly renders `:param dict(str, str) opc_meta:` because `fieldarg.split(None, 1)` in `sphinx/util/docfields.py` splits on the first whitespace *inside* the parenthesized type, misaligning the type–name boundary.

Mini-SWE-Agent (117 steps). Without pre-repair knowledge, the agent is misled by surface-level keyword overlap. It patches a superficially similar split in `typehints.py`, then pivots to `writers/html.py` and `autodoc/__init__.py` after each attempt fails testing, cycling through increasingly ad-hoc workarounds across four files. The correct file `docfields.py` is never visited.

ACQUIRE (52 steps). We describe how each stage of ACQUIRE contributes to solving this issue.

Stage 1: Knowledge Acquisition. The Questioner generates a question targeting the *parsing mechanism* of `:param` type extraction, rather than the rendering pipeline. This directs exploration toward the causal logic of how types are parsed, enabling discovery of utility-layer code that shares no surface lexical overlap with the issue description. The Answerer first examines keyword-related modules such as `autodoc/` and `napoleon/`, but finds that neither contains the actual `:param` type-parsing logic. It then broadens its search to `sphinx/util/` and discovers `docfields.py`, pinpointing the root cause: the split breaks when spaces appear inside parenthesized types. The answer provides the exact file path, class name, line number, and root-cause explanation.

Stage 2: Knowledge-Informed Repair. With the acquired QA knowledge, the Resolver directly opens `docfields.py`, confirms the problematic `split` call, and implements a bracket-aware replacement. The resulting patch modifies exactly one file with a minimal, focused change. Notably, the agent designs comprehensive test cases covering diverse compound-typed `:param` directives to verify the fix, and the final patch directly addresses the root cause identified in the QA answer, echoing both the behavioral shift toward verification and the strong QA–patch alignment observed in Finding 2. The entire trajectory completes in **52 steps**, a **55%** reduction from the baseline's 117 steps.

Analysis. Beyond Mini-SWE-Agent, we also inspect the pre-repair exploration outputs of other baselines: none surfaces the root-cause file `docfields.py`, as the issue description centers on rendering symptoms rather than the underlying parsing logic, misleading keyword-driven exploration away from the actual bug site. This case highlights two advantages

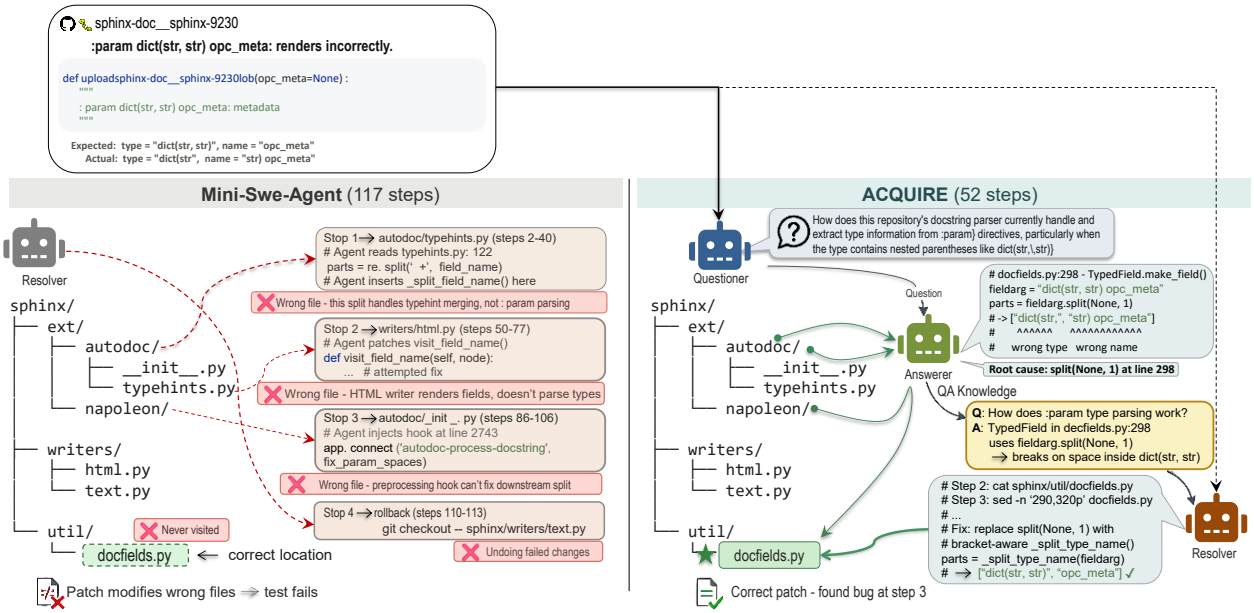

Fig. 4. Case Study of ACQUIRE with instance sphinx-doc__sphinx-9230.

of ACQUIRE. First, by decomposing the issue into a targeted question, ACQUIRE decouples knowledge acquisition from repair and directs the Answerer to investigate the underlying mechanism rather than chase surface-level keyword matches, yielding a precise and well-grounded answer. Second, the semantically rich QA knowledge allows the repair agent to inherit accurate fault localization and root-cause understanding, effectively offloading a significant portion of the exploratory burden and substantially improving both efficiency and repair capability.

VI. Discussion

A. Strengths

Our study demonstrates that explicitly acquiring repository knowledge before repair is practical and effective for repository-level issue resolution. By decoupling question generation, repository-grounded answering, and patch synthesis, ACQUIRE turns implicit knowledge gaps into explicit QA knowledge before editing begins. This design directly addresses the recurring knowledge-deficit failure patterns identified in Section II, and operationalizes a principle long observed in empirical debugging research: expert developers consistently invest more effort in program comprehension before attempting repairs, yielding faster and more accurate outcomes than jumping directly into code modification [14], [15].

Empirically, ACQUIRE shows consistent gains across different backbone models at modest additional cost. The improvements are not only reflected in Pass@1, but also in behavioral efficiency: the repair agent spends less effort on blind locating and trial-and-error fixing, and more on reproduction and verification. This indicates that pre-repair knowledge acquisition improves both effectiveness and the quality of repair trajectories.

Another strength is interpretability. The category-guided question template provides a clear interface for what knowledge is being requested, while the QA pairs make it easier to inspect, diagnose, and analyze why a repair succeeds or fails. Compared with the localization lists or repair descriptions produced by existing pre-repair methods, this structured QA pipeline offers better controllability and clearer debugging signals for future system improvement.

B. Limitations and Future Work

Despite these strengths, the current framework still has room for further improvement.

The four-category question template, while practical and effective, can be further refined for broader coverage. Future work can explore hierarchical categories, issue-type-adaptive templates, or automatically learned category schemas from large-scale repair trajectories.

Additionally, knowledge is currently injected as pre-generated QA context before repair, which has proven simple and stable in practice. A promising extension is to couple knowledge acquisition with the repair trajectory itself, allowing the agent to dynamically request and refresh knowledge as new hypotheses emerge during debugging. However, dynamic acquisition would inevitably introduce additional cost; exploring how to balance knowledge freshness and computational cost remains an open challenge.

VII. Threats to Validity

We discuss potential validity threats from three perspectives.

Construct validity. A key threat is whether our metrics fully capture repair quality. We use Pass@1 on SWE-bench Verified as the primary effectiveness metric, and complement it with time, cost, and trajectory-step analyses that provide an efficiency perspective on the repair process.

Internal validity. Observed gains could be influenced by confounding factors such as prompt design, tool behavior, or hyperparameter choices rather than the proposed QA-driven mechanism itself. To reduce this risk, we keep the repair backbone and evaluation protocol consistent across methods, and conduct targeted ablations that remove core components (question decomposition and category-guided question generation). The consistent performance drops in these ablations support the causal contribution of our key design choices. Additionally, the question-quality evaluation in RQ3 relies on LLM-as-a-judge; to mitigate potential scoring bias, we adopt both scoring and pairwise voting perspectives, repeat each judgment 10 times, and apply position-bias mitigation following established practices [27].

External validity. Our current prompt design and evaluation are tailored to Python repositories. While the QA-driven paradigm itself is language-agnostic, adapting the prompt design to other programming languages and validating its effectiveness across them remains future work.

VIII. Related Work

A. Software Issue Resolution

Repository-level issue resolution addresses real-world software bugs by reasoning over cross-file dependencies and generating patches within full repositories. Unlike function-level repair, it demands broader context understanding and robust handling of ambiguous descriptions. Recent progress is driven by the SWE-bench benchmarks [13], [19], [32] and supporting infrastructure such as Repo2Run for scalable environment construction [33], while agentic frameworks like SWE-agent enable models to interact with realistic software environments [4]. Alongside these advances, a growing body of work scrutinizes benchmark reliability, examining issues such as test overfitting, label quality, evaluation inflation, and benchmark extensibility [34]–[41].

Existing approaches follow three main directions. First, structured workflow decomposition improves controllability and reasoning. For instance, Agentless splits repair into localization, repair, and validation stages [5]. AutoCodeRover and LingmaAgent enhance repository search and multi-step reasoning [6], [12], while SWE-Debate introduces structured debate into the repair pipeline [1].

Second, effective repository exploration and context management are critical. LocAgent uses graph-guided multi-hop reasoning for localization [3], and RepoGraph leverages structural representations for better traversal [42]. Additionally, EET and SWE-Pruner emphasize efficiency control and context pruning to streamline search and reading processes [2], [21], and recent work further explores compressing code context to reduce input length while preserving repair-relevant information [43].

Third, leveraging reusable knowledge and memory enhances agentic software engineering [44]. Tools like SWE-Exp, EXPEREPAIR, and Agent KB utilize historical experience, dual-memory designs, and shared knowledge bases to inform repair decisions [45]–[47]. Similarly, SAGE and SE-Agent reuse execution trajectories for self-evolution and planning [48], [49].

Empirical studies further illuminate systematic failure patterns in automated issue resolution [7], [8], [50]. These analyses identify knowledge deficits, particularly insufficient understanding of repository internals and behavioral contracts, as a primary source of incorrect or incomplete patches, motivating explicit pre-repair knowledge acquisition. Despite these advances, prior methods assume that existing mechanisms can surface all relevant evidence, neglecting explicit knowledge-gap identification before patch synthesis [1]–[6], [12], [21], [42], [45]–[49]. ACQUIRE overcomes this limitation by decoupling question generation, repository-grounded answering, and patch synthesis, enabling proactive acquisition of missing contextual evidence prior to repair.

B. Repository QA and Knowledge Acquisition

Repository-level QA has evolved from semantic code search benchmarks such as CodeSearchNet and CoSQA+, which align natural-language intents with code snippets [51], [52]. Modern efforts scale this to full repositories, requiring multi-file traversal and evidence aggregation across multiple interdependent files.

Recent benchmarks formalize repository-scale comprehension. RepoQA evaluates long-context code understanding [53], while CodeRepoQA, CoReQA, and SWE-QA introduce large-scale QA datasets derived from software engineering tasks, GitHub issues, and complex multi-hop dependencies [16], [54], [55]. These benchmarks reveal that repository-level QA demands not only retrieval precision but also the ability to synthesize answers from evidence scattered across multiple abstraction layers.

Repository QA increasingly intersects with agentic reasoning and structured retrieval. Tools like RepoGraph enhance navigation via structural representations [42], while SWE-agent and RepoCoder utilize iterative retrieval and generation for interactions and code completion [4], [56]. Furthermore, RepoQA-Agent shifts repository QA from static retrieval to interactive, search-driven answering guided by reinforcement learning [57].

Despite these advancements, existing QA frameworks typically assume a human-to-repository paradigm where the question is pre-provided, focusing solely on retrieval and answer generation [16], [42], [53]–[55], [57]. ACQUIRE shifts this to an agent-to-repository model. Rather than treating QA as an external benchmark, ACQUIRE integrates it as an internal capability, enabling the agent to autonomously identify knowledge gaps, formulate questions, and acquire contextual evidence prior to patch generation.

IX. Conclusion

This paper presented ACQUIRE, a QA-driven framework for repository-level software issue resolution. Instead of directly searching and patching from issue keywords, ACQUIRE explicitly identifies missing knowledge, acquires repository-grounded answers, and then performs repair with this structured evidence.

Across experiments on SWE-bench Verified, ACQUIRE consistently improves issue resolution over strong baselines at modest additional cost and remains robust under different model backbones. Behavioral analysis further reveals that QA injection accelerates knowledge-intensive repair stages and shifts agent effort toward verification, with the acquired knowledge ultimately encoded into the majority of successful patches. Ablation studies confirm that the gains stem from two key design choices: decomposing issues into targeted, answerable questions and using category-guided question generation to obtain more diagnostic and complementary knowledge.

Overall, our results suggest that explicit pre-repair knowledge acquisition is a critical ingredient for reliable agentic software engineering, and ACQUIRE provides a practical path toward more accurate, efficient, and interpretable automated issue resolution.

Data Availability

All code and data used in this study are publicly available at: https://github.com/LionLin2003/ACQUIRE.

References

[1] H. Li, Y. Shi, S. Lin, X. Gu, H. Lian, X. Wang, Y. Jia, T. Huang, and Q. Wang, "Swe-debate: Competitive multi-agent debate for software issue resolution," *arXiv preprint arXiv:2507.23348*, 2025.

[2] Y. Wang, Y. Shi, M. Yang, R. Zhang, S. He, H. Lian, Y. Chen, S. Ye, K. Cai, and X. Gu, "Swe-pruner: Self-adaptive context pruning for coding agents," *arXiv preprint arXiv:2601.16746*, 2026.

[3] Z. Chen, X. Tang, G. Deng, F. Wu, J. Wu, Z. Jiang, V. Prasanna, A. Cohan, and X. Wang, "Locagent: Graph-guided llm agents for code localization," in *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Vienna, Austria: Association for Computational Linguistics, jul 2025, pp. 8697–8727. [Online]. Available: https://aclanthology.org/2025.acl-long.426/

[4] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. R. Narasimhan, and O. Press, "SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering," in *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, Nov. 2024.

[5] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang, "Agentless: Demystifying LLM-based software engineering agents," 2024. [Online]. Available: https://arxiv.org/abs/2407.01489

[6] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "Autocoderover: Autonomous program improvement," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024.

[7] S. Liu, F. Liu, L. Li, X. Tan, Y. Zhu, X. Lian, and L. Zhang, "An empirical study on failures in automated issue solving," September 2025. [Online]. Available: https://arxiv.org/abs/2509.13941

[8] I. Bouzenia and M. Pradel, "Understanding software engineering agents: A study of thought-action-result trajectories," in *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering*, 2025.

[9] K. He and K. Roy, "SWE-Adept: An LLM-based agentic framework for deep codebase analysis and structured issue resolution," March 2026. [Online]. Available: https://arxiv.org/abs/2603.01327

[10] Z. Fan, K. Vasilevski, D. Lin, B. Chen, Y. Chen, Z. Zhong, J. M. Zhang, P. He, and A. E. Hassan, "SWE-Effi: Re-evaluating software AI agent system effectiveness under resource constraints," September 2025. [Online]. Available: https://arxiv.org/abs/2509.09853

[11] Z. Jiang, X. Ren, M. Yan, W. Jiang, Y. Li, and Z. Liu, "Issue localization via llm-driven iterative code graph searching," in *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering*, 2025, pp. 3034–3045.

[12] Y. Ma, Q. Yang, R. Cao, B. Li, F. Huang, and Y. Li, "Alibaba LingmaAgent: Improving automated issue resolution via comprehensive repository exploration," in *Companion Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion '25)*. New York, NY, USA: Association for Computing Machinery, 2025, pp. 238–249.

[13] OpenAI, "Swe-bench verified," https://openai.com/index/introducing-swe-bench-verified/, 2024.

[14] I. Vessey, "Expertise in debugging computer programs: A process analysis," *International Journal of Man-Machine Studies*, vol. 23, no. 5, pp. 459–494, 1985.

[15] L. Gugerty and G. M. Olson, "Debugging by skilled and novice programmers," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1986, pp. 171–174.

[16] W. Peng, Y. Shi, Y. Wang, X. Zhang, B. Shen, and X. Gu, "Swe-qa: Can language models answer repository-level code questions?" *arXiv preprint arXiv:2509.14635*, 2025.

[17] S. Vijayvargiya, X. Zhou, A. Yerukola, M. Sap, and G. Neubig, "Ambigswe: Interactive agents to overcome underspecificity in software engineering," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2026.

[18] J. Yang, C. E. Jimenez, K. Lieret, A. Wettig, S. Yao, K. Narasimhan, and O. Press, "mini-SWE-agent: The minimal AI software engineering agent," GitHub repository, 2024, accessed: 2026-03-26. [Online]. Available: https://github.com/SWE-agent/mini-swe-agent

[19] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. R. Narasimhan, "Swe-bench: Can language models resolve real-world github issues?" in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2024.

[20] DeepSeek-AI, A. Liu, A. Mei, B. Lin *et al.*, "Deepseek-v3.2: Pushing the frontier of open large language models," *arXiv preprint arXiv:2512.02556*, 2025.

[21] Y. Guo, Y. Xiao, J. M. Zhang, M. Harman, Y. Lou, Y. Liu, and Z. Chen, "Eet: Experience-driven early termination for cost-efficient software engineering agents," *arXiv preprint arXiv:2601.05777*, 2026.

[22] M. Suri, X. Li, M. Shojaie, S. Han, C.-C. Hsu, S. Garg, A. A. Deshmukh, and V. Kumar, "Codescout: Contextual problem statement enhancement for software agents," *arXiv preprint arXiv:2603.05744*, 2026.

[23] C. S. Xia, Z. Wang, Y. Yang, Y. Wei, and L. Zhang, "Live-swe-agent: Can software engineering agents self-evolve on the fly?" *arXiv preprint arXiv:2511.13646*, 2025.

[24] H. Hayashi, B. Pang, W. Zhao, Y. Liu, A. Gokul, S. Bansal, C. Xiong, S. Yavuz, and Y. Zhou, "Self-abstraction from grounded experience for plan-guided policy refinement," November 2025. [Online]. Available: https://arxiv.org/abs/2511.05931

[25] A. Singh, A. Fry, A. Perelman, A. Tart, A. Ganesh, A. El-Kishky, A. McLaughlin *et al.*, "OpenAI GPT-5 system card," 2025, technical Report by OpenAI. [Online]. Available: https://arxiv.org/abs/2601.03267

[26] J. Gu, X. Jiang, Z. Shi, H. Tan, X. Zhai, C. Xu, W. Li, Y. Shen, S. Ma, H. Liu, S. Wang, K. Zhang, Y. Wang, W. Gao, L. Ni, and J. Guo, "A survey on LLM-as-a-judge," *The Innovation*, vol. 7, p. 101253, 2026.

[27] L. Shi, C. Ma, W. Liang, X. Diao, W. Ma, and S. Vosoughi, "Judging the judges: A systematic study of position bias in LLM-as-a-judge," in *Proceedings of the 14th International Joint Conference on Natural Language Processing and the 4th Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics*. Mumbai, India: The Asian Federation of Natural Language Processing and The Association for Computational Linguistics, dec 2025, pp. 292–314. [Online]. Available: https://aclanthology.org/2025.ijcnlp-long.18/

[28] T. Tripathi, M. Wadhwa, G. Durrett, and S. Niekum, "Pairwise or pointwise? evaluating feedback protocols for bias in llm-based evaluation," in *Proceedings of the Conference on Language Modeling (COLM)*, 2025.

[29] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, "Lost in the middle: How language models use long contexts," *Transactions of the Association for Computational Linguistics*, vol. 12, pp. 157–173, 2024.

[30] F. Shi, X. Chen, K. Misra, N. Scales, D. Dohan, E. H. Chi, N. Schärli, and D. Zhou, "Large language models can be easily distracted by irrelevant context," in *Proceedings of the 40th International Conference on Machine Learning*. PMLR, 2023, pp. 31 210–31 227.

[31] M. Yang, E. Huang, L. Zhang, M. Surdeanu, W. Y. Wang, and L. Pan, "How is LLM reasoning distracted by irrelevant context? an analysis using a controlled benchmark," in *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*. Suzhou, China:

Association for Computational Linguistics, nov 2025, pp. 13 329–13 347. [Online]. Available: https://aclanthology.org/2025.emnlp-main.674/

[32] D. Ma, S. Chen, Y. Yang, Y. Shi, Y. Yan, and X. Gu, "Llm agents can see code repositories," 2026. [Online]. Available: https://arxiv.org/abs/2606.14061

[33] R. Hu, C. Peng, X. Wang, J. Xu, and C. Gao, "Repo2Run: Automated building executable environment for code repository at scale," *arXiv preprint arXiv:2502.13681*, 2025. [Online]. Available: https://arxiv.org/abs/2502.13681

[34] Y. Wang, M. Pradel, and Z. Liu, "Are "solved issues" in SWE-bench really solved correctly? an empirical study," *arXiv preprint arXiv:2503.15223*, 2025. [Online]. Available: https://arxiv.org/abs/2503.15223

[35] T. Ahmed, J. Ganhotra, A. Shinnar, and M. Hirzel, "Investigating test overfitting on SWE-bench," *arXiv preprint arXiv:2511.16858*, 2025. [Online]. Available: https://arxiv.org/abs/2511.16858

[36] B. Yu, Y. Cao, Y. Zhang, L. Lin, J. Xu, Z. Zhong, Q. Xu, G. Wang, J. Cao, S.-C. Cheung, P. He, and L. Briand, "SWE-ABS: Adversarial benchmark strengthening exposes inflated success rates on test-based benchmark," *arXiv preprint arXiv:2603.00520*, 2026. [Online]. Available: https://arxiv.org/abs/2603.00520

[37] G. A. Oliva, G. K. Rajbahadur, A. Bhatia, H. Zhang, Y. Chen, Z. Chen, A. Leung, D. Lin, B. Chen, and A. E. Hassan, "SPICE: An automated SWE-Bench labeling pipeline for issue clarity, test coverage, and effort estimation," *arXiv preprint arXiv:2507.09108*, 2025. [Online]. Available: https://arxiv.org/abs/2507.09108

[38] Y. Chen, T. Ahmed, R. Jabbarvand, and M. Hirzel, "Can old tests do new tricks for resolving SWE issues?" *arXiv preprint arXiv:2510.18270*, 2025. [Online]. Available: https://arxiv.org/abs/2510.18270

[39] B. Yu, Y. Zhu, P. He, and D. Kang, "UTBoost: Rigorous evaluation of coding agents on SWE-Bench," *arXiv preprint arXiv:2506.09289*, 2025. [Online]. Available: https://arxiv.org/abs/2506.09289

[40] L. Wang, L. Ramalho, A. Celestino, P. A. Pham, Y. Liu, U. K. Sinha, A. Portillo, O. Osunwa, and G. Maduekwe, "SWE-Bench++: A framework for the scalable generation of software engineering benchmarks from open-source repositories," *arXiv preprint arXiv:2512.17419*, 2025. [Online]. Available: https://arxiv.org/abs/2512.17419

[41] S. Garg, B. Steenhoek, and Y. Huang, "Saving SWE-Bench: A benchmark mutation approach for realistic agent evaluation," *arXiv preprint arXiv:2510.08996*, 2025, accepted at CAIN 2026. [Online]. Available: https://arxiv.org/abs/2510.08996

[42] S. Ouyang, W. Yu, K. Ma, Z. Xiao, Z. Zhang, M. Jia, J. Han, H. Zhang, and D. Yu, "Repograph: Enhancing ai software engineering with repository-level code graph," in *13th International Conference on Learning Representations, ICLR 2025*. International Conference on Learning Representations, ICLR, 2025, pp. 30 361–30 384.

[43] H. Jia, E. T. Barr, and S. Mechtaev, "Compressing code context for LLM-based issue resolution," *arXiv preprint arXiv:2603.28119*, 2026. [Online]. Available: https://arxiv.org/abs/2603.28119

[44] Y. Lin, Y. Ma, R. Cao, B. Li, F. Huang, X. Gu, and Y. Li, "Llms as continuous learners: Improving the reproduction of defective code in software issues," *arXiv preprint arXiv:2411.13941*, 2024.

[45] S. Chen, S. Lin, X. Gu, Y. Shi, H. Lian, L. Yun, D. Chen, W. Sun, L. Cao, and Q. Wang, "Swe-exp: Experience-driven software issue resolution," *arXiv preprint arXiv:2507.23361*, 2025.

[46] F. Mu, J. Wang, L. Shi, S. Wang, S. Li, and Q. Wang, "Experepair: Dual-memory enhanced llm-based repository-level program repair," *arXiv preprint arXiv:2506.10484*, 2025.

[47] X. Tang, T. Qin, T. Peng, Z. Zhou, D. Shao, T. Du, X. Wei, P. Xia, F. Wu, H. Zhu, G. Zhang, J. Liu, X. Wang, S. Hong, C. Wu, H. Cheng, C. Wang, and W. Zhou, "Agent kb: Leveraging cross-domain experience for agentic problem solving," *arXiv preprint arXiv:2507.06229*, 2025.

[48] H. Hayashi, B. Pang, W. Zhao, Y. Liu, A. Gokul, S. Bansal, C. Xiong, S. Yavuz, and Y. Zhou, "Self-abstraction from grounded experience for plan-guided policy refinement," *arXiv preprint arXiv:2511.05931*, 2025.

[49] J. Lin, Y. Guo, Y. Han, S. Hu, Z. Ni, L. Wang, M. Chen, H. Liu, R. Chen, Y. He *et al.*, "Se-agent: Self-evolution trajectory optimization in multi-step reasoning with llm-based agents," *arXiv preprint arXiv:2508.02085*, 2025.

[50] W. Ma, Z. Chen, J. Gu, T. Li, S. Liu, and L. Jiang, "Same signal, different semantics: A cross-framework behavioral analysis of software engineering agents," *arXiv preprint arXiv:2605.18332*, 2026. [Online]. Available: https://arxiv.org/abs/2605.18332

[51] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[52] J. Gong, Y. Wu, L. Liang, Y. Wang, J. Chen, and M. Liu, "Cosqa+: Enhancing code search evaluation with a multi-choice benchmark and test-driven agents," *IEEE Transactions on Software Engineering*, vol. 52, no. 1, pp. 206–220, 2026.

[53] J. Liu, J. L. Tian, V. Daita, Y. Wei, Y. Ding, Y. K. Wang, J. Yang, and L. Zhang, "Repoqa: Evaluating long context code understanding," *arXiv preprint arXiv:2406.06025*, 2024.

[54] R. Hu, C. Peng, J. Ren, B. Jiang, X. Meng, Q. Wu, P. Gao, X. Wang, and C. Gao, "Coderepoqa: A large-scale benchmark for software engineering question answering," *arXiv preprint arXiv:2412.14764*, 2024.

[55] J. Chen, K. Zhao, J. Liu, C. Peng, J. Liu, H. Zhu, P. Gao, P. Yang, and S. Deng, "Coreqa: Uncovering potentials of language models in code repository question answering," *arXiv preprint arXiv:2501.03447*, 2025.

[56] F. Zhang, B. Chen, Y. Zhang, J. Keung, J. Liu, D. Zan, Y. Mao, J.-G. Lou, and W. Chen, "Repocoder: Repository-level code completion through iterative retrieval and generation," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. Singapore: Association for Computational Linguistics, dec 2023, pp. 2471–2484. [Online]. Available: https://aclanthology.org/2023.emnlp-main.151/

[57] G. Li, Y. Liu, Z. Qin, Y. Wang, J. Zhong, C. Zhi, B. Li, F. Huang, Y. Li, and S. Deng, "Empowering repoqa-agent based on reinforcement learning driven by monte-carlo tree search," *arXiv preprint arXiv:2510.26287*, 2025.

Supplementary Material

I. Prompt Templates

A. Questioner Prompt

System Prompt

```
You are an expert software engineer tasked
with generating questions to help resolve
GitHub issues.

# Task Overview
Generate exactly {N} questions that are most
relevant to resolving the given issue. These
questions should focus on understanding the
repository's current state and implementation.

# Question Categories
Select from the following 4 categories based
on the issue content. Categories can be
repeated if both top questions fall into the
same category.

## 1. Mechanism & Behavior
Questions about specific functional logic
flows, state management, data processing
details, and internal workings of specific
components. Questions that ask "**how does
this functionality actually work**".

## 2. Design & Usage
Questions about API definitions, class
inheritance hierarchies, error specifications,
 design pattern applications, and type safety
design. Questions that ask "**how interfaces
are designed and what contracts they follow
**".

## 3. Locating & Structure
Questions about the codebase's macro layout,
module division, dependency relationships, and
 file location knowledge. Questions that ask
"**where things are and what the overall
structure looks like**".

## 4. Ecosystem & Standards
Questions about external technical ecosystems
and general rules that the project depends on,
 including third-party libraries, programming
language features, official standard libraries
, industry protocols, file format standards,
and mathematical/scientific concepts.
Questions that ask "**what external knowledge
is needed**".

Note: Categories 1-3 are repository knowledge
that can be obtained from within the project.
Category 4 is external knowledge independent
of the repository.

# Guidelines

## DO:
- Focus on understanding the repository's
current implementation
- Ask questions whose answers can be obtained
by analyzing the code repository
- Ask about the current state of the code as
it exists now
- Prioritize questions by importance (Q1 is
most important, Q2 is second)
- Use generic terms like "this repository", "
the codebase", "this project" instead of
specific project names

## DO NOT:
- Ask about issue-specific details that are
only in the issue description
- Ask about external knowledge or resources
outside the repository (unless for Category 4)
- Ask about code changes, git operations,
commits, branches, or versions
- **NEVER reference specific commit hashes,
commit numbers, or version changes**
- **NEVER ask about "changes introduced in
commit X" or "what was modified in PR Y"**
- **NEVER use specific project names - use "
this repository", "the codebase", "this
project" instead**
- Generate duplicate or highly similar
questions

# Output Format
Return ONLY a valid JSON object with the
following structure:

'''json
{
  "Q1": {
    "category": "Mechanism & Behavior",
    "question": "How does the authentication
    system validate user credentials in this
    repository?"
  },
  "Q2": {
    "category": "Locating & Structure",
    "question": "Where is the login
    functionality implemented in the codebase
    ?"
  }
}
'''

Important:
- Generate exactly {N} questions
- Questions should be sorted by importance (Q1
 being most important)
- Each question must have both "category" and
"question" fields
- Use the exact category names provided above
- Categories can repeat if both questions
belong to the same category
- Do not include any text outside the JSON
object
```

User Prompt

```
Here is the GitHub issue problem statement:

{problem_statement}

Please generate exactly {N} questions based on
 the above problem statement according to the
specified requirements and output format.
```

B. Answerer Instance Template

Instance Template

````
<question>
{{task}}
</question>

<instructions>
# Task Instructions

## Overview
You're a software engineer analyzing a
codebase by interacting with a computer shell.
Your task is to understand the repository and
answer the question provided above.

IMPORTANT: This is a READ-ONLY task. You
should NOT modify any files in the repository.
Your goal is to explore, read, and analyze the
 code to answer the question accurately.

For each response:
1. Include a THOUGHT section explaining your
reasoning and what you're trying to accomplish
2. Provide exactly ONE bash command to execute

## Important Rules
- READ ONLY: You can read files, search code,
list directories, run analysis commands
- DO NOT MODIFY: Do not create, edit, or
delete any files in /testbed
- DO NOT RUN: Avoid running tests or executing
 application code unless necessary for
understanding

## Recommended Workflow
1. Understand the question thoroughly
2. Explore the repository structure to locate
relevant code
3. Read and analyze relevant files
4. Search for patterns, function definitions,
or specific implementations
5. Synthesize information to form your answer
6. Submit your final answer

## Command Execution Rules
You are operating in an environment where:
1. You write a single command
2. The system executes that command in a
subshell
3. You see the result
4. You write your next command

Each response should include:
1. A **THOUGHT** section where you explain
your reasoning and plan
2. A single bash code block with your command

Format your responses like this:

<format_example>
THOUGHT: Here I explain my reasoning process,
analysis of the current situation,
and what I'm trying to accomplish with the
command below.

```bash
your_command_here
```
</format_example>

Commands must be specified in a single bash
code block:

```bash
your_command_here
```

**CRITICAL REQUIREMENTS:**
- Your response SHOULD include a THOUGHT
section explaining your reasoning
- Your response MUST include EXACTLY ONE bash
code block
- This bash block MUST contain EXACTLY ONE
command (or a set of commands connected with
&& or ||)
- If you include zero or multiple bash blocks,
 or no command at all, YOUR RESPONSE WILL FAIL
- Do NOT try to run multiple independent
commands in separate blocks in one response
- Directory or environment variable changes
are not persistent. Every action is executed
in a new subshell.
- However, you can prefix any action with `
MY_ENV_VAR=MY_VALUE cd /path/to/working/dir &&
 ...` or write/load environment variables from
 files

Example of a CORRECT response:
<example_response>
THOUGHT: I need to understand the structure of
 the repository first. Let me check what files
 are in the current directory to get a better
understanding of the codebase.

```bash
ls -la
```
</example_response>

Example of an INCORRECT response:
<example_response>
THOUGHT: I need to examine the codebase and
then look at a specific file. I'll run
multiple commands to do this.

```bash
ls -la
```

Now I'll read the file:

```bash
cat file.txt
```
</example_response>

If you need to run multiple commands, either:
1. Combine them in one block using && or ||
```bash
command1 && command2 || echo "Error occurred"
```

2. Wait for the first command to complete, see
 its output, then issue the next command in
your following response.

## Environment Details
- You have a full Linux shell environment
- Always use non-interactive flags (-y, -f)
for commands
- Avoid interactive tools like vi, nano, or
any that require user input
````

```
- All standard Unix tools are available (grep,
 find, cat, head, tail, etc.)

## Useful Command Examples for Code Analysis

### List repository structure:
```bash
find /testbed -type f -name "*.py" | head -20
```

Search for specific code patterns:
```bash
grep -r "function_name" /testbed --include="*.
py"
```

View file content:
```bash
cat /testbed/path/to/file.py
```

View specific lines with numbers:
```bash
nl -ba filename.py | sed -n '10,20p'
```

Search with context:
```bash
grep -A 5 -B 5 "pattern" filename.py
```

Find file by name:
```bash
find /testbed -name "models.py"
```

Count lines or analyze code:
```bash
wc -l /testbed/**/*.py
```

Answer Submission
When you have gathered enough information and
are ready to provide your final answer,
issue exactly the following command:

```bash
cat <<'ANSWER_EOF'
SUBMIT_ANSWER
Your detailed answer here. Be specific,
thorough, and reference the code/files you
analyzed.
Include relevant file paths, function names,
and explanations.
ANSWER_EOF
```

The format is CRITICAL:
- First line after opening must be:
SUBMIT_ANSWER
- Following lines contain your answer
- Use the heredoc syntax exactly as shown
above

This command will submit your answer.
You cannot continue working on this task after
 submitting.
</instructions>
```

C. Question-Quality Evaluation Prompts

1) Scoring – Individual Question:

System Prompt

```
You are a rigorous evaluator assessing the
quality of a single repository-focused
diagnostic question generated from a GitHub
issue.

Evaluate the question only using the provided
problem statement and the question text itself
.

A strong question should be tightly grounded
in the repository, highly relevant to
resolving the issue, diagnostically useful for
 narrowing the debugging search space, and
clearly phrased.
```

User Prompt

```
## Input

Problem Statement:
{problem_statement}

Question to Evaluate:
{question}

## Task

Evaluate the single question across the five
dimensions below.
Use integer scores from 1 to 10.

## Scoring Dimensions

1. Relevance
- 1-2: Irrelevant to the issue's core failure
or likely fix path.
- 3-4: Weakly related but mostly about side
details or symptoms.
- 5-6: Moderately relevant but still broad or
indirect.
- 7-8: Strongly relevant to understanding or
resolving the issue.
- 9-10: Precisely targets the likely failure
mechanism, root cause, or key repair decision.

2. Repository Answerability
- 1-2: Requires mainly outside knowledge or
general internet or documentation lookup.
- 3-4: Only loosely tied to the repository;
answerable mostly with generic software
reasoning.
- 5-6: Partly grounded in repository artifacts
 but still broad.
- 7-8: Clearly answerable through repository
code, tests, config, or docs.
- 9-10: Strongly grounded in specific internal
 implementation, control flow, state
propagation, validation logic, or repository-
specific architecture.

3. Diagnostic Utility
- 1-2: Adds almost no debugging value.
- 3-4: Some value, but leaves the search space
 very large.
- 5-6: Moderately useful for narrowing the
investigation.
- 7-8: Strongly helps isolate a subsystem,
```

```
state transition, boundary condition, or
execution path.
- 9-10: Highly discriminative; answering it
would sharply narrow the fault to a specific
mechanism, guard condition, state transition,
or repair decision.

4. Reasoning Depth
- 1-2: Trivial lookup or superficial
restatement.
- 3-4: Simple one-file or one-symbol lookup.
- 5-6: Requires some local reasoning over
behavior or flow.
- 7-8: Requires tracing meaningful
interactions across functions, files, or
states.
- 9-10: Requires substantial multi-hop
reasoning across decoupled components, state
changes, or abstraction boundaries.

5. Clarity
- 1-2: Ambiguous, confusing, or combines
multiple unrelated asks.
- 3-4: Understandable but poorly scoped or
imprecise.
- 5-6: Acceptable but somewhat verbose or
fuzzy.
- 7-8: Clear, well-scoped, and technically
precise.
- 9-10: Very concise, unambiguous, and sharply
 targeted.

## Hard Flags

Set each to true or false.

- requires_external_knowledge: answering the
question fundamentally depends on external
domain knowledge rather than repository
artifacts.
- too_generic: the question is generic enough
to fit almost any software issue.
- rephrases_issue_without_added_value: the
question mostly restates the issue without
adding diagnostic direction.

## Guidance

Repository Answerability is about where the
answer comes from.
Diagnostic Utility is about how much the
answer would reduce debugging uncertainty.
Do not conflate them.

A question that could be answered reasonably
well without opening the repository should not
 receive a high Repository Answerability score
.
A question that merely suggests where to start
 looking, without meaningfully reducing
uncertainty, should not receive a high
Diagnostic Utility score.

Do not reward complexity for its own sake.
A focused question may deserve a high score
even if it does not require broad multi-file
reasoning.

## Calibration Examples

Example Good:
- Problem: "Parser crashes when optional
config is missing."
- Question: "Where is the default config
object created, and which call path assumes it
 is non-null before validation?"

Example Bad:
- Problem: "Build fails on Python 3.12."
- Question: "What are the exact reproduction
steps and environment details?"

Example Weak Repository Question:
- Problem: "A validation error appears in one
edge case."
- Question: "Why does this issue happen in the
 application?"
- Why weak: broad, generic, and not clearly
answerable from repository internals.

## Output Format

Return only valid JSON.

{
  "scores": {
    "relevance": 1,
    "repository_answerability": 1,
    "diagnostic_utility": 1,
    "reasoning_depth": 1,
    "clarity": 1
  },
  "hard_flags": {
    "requires_external_knowledge": false,
    "too_generic": false,
    "rephrases_issue_without_added_value":
    false
  },
  "summary": "One concise sentence explaining
  the judgment.",
  "dimension_rationale": {
    "relevance": "Short explanation.",
    "repository_answerability": "Short
    explanation.",
    "diagnostic_utility": "Short explanation
    .",
    "reasoning_depth": "Short explanation.",
    "clarity": "Short explanation."
  }
}
```

2) Scoring – Question Set (Coverage):

System Prompt

```
You are an expert evaluator assessing the
quality of a set of repository-focused
diagnostic questions generated from a GitHub
issue.

Evaluate the set only using the provided
problem statement and the question texts
themselves.

A strong question set should efficiently cover
 the key diagnostic angles needed to
understand, localize, and resolve the issue.
High-quality coverage means the questions are
collectively relevant to the core defect,
complementary rather than repetitive, and
useful for narrowing the debugging search
space.
```

User Prompt

```
## Input

Problem Statement:
{problem_statement}

Questions:
{question_set}

## Task

Evaluate the question set as a whole using a
single dimension: Coverage.
Use an integer score from 1 to 10, where 10 is
 always the best score.

## Scoring Dimension

1. Coverage
Coverage measures how well this set uses its
limited question slots to cover the key
diagnostic angles needed to debug this
specific issue.

When scoring coverage, consider all of the
following together:
- Core relevance: Are the questions
collectively centered on the main failure
mechanism rather than peripheral details?
- Complementarity: Do the questions probe
different useful diagnostic angles rather than
 circling the same point?
- Non-redundancy: Does each question add
distinct information value, instead of
rephrasing another question?
- Debugging usefulness: Would answering this
set significantly reduce uncertainty about
where and why the bug occurs?

Behavioral anchors:
- 1-2: Very poor coverage. The set is generic,
 repetitive, off-target, or misses the key
diagnostic needs of the issue.
- 3-4: Weak coverage. Some questions are
related, but the set leaves major diagnostic
gaps and wastes question slots on overlap or
shallow prompts.
- 5-6: Moderate coverage. The set addresses
part of the debugging space, but misses
important complementary angles or contains
noticeable redundancy.
- 7-8: Strong coverage. The set covers several
 important and complementary diagnostic angles
 with mostly useful and non-redundant
questions.
- 9-10: Excellent coverage. The set
efficiently covers the most important
diagnostic angles needed for this issue, with
highly complementary, targeted, and
information-dense questions.

## Guidance

Do not reward breadth or question count for
their own sake.
A smaller set can score higher if it covers
the key diagnostic needs more efficiently.
Reworded versions of the same underlying
question should not be treated as additional
coverage.
Coverage should reflect usefulness for
debugging this specific issue, not general
interestingness.

## Calibration Examples

Example Better Set:
- Q1 identifies the likely failing file,
subsystem, or execution path.
- Q2 asks how the relevant control flow or
state transition currently works.
- Why better: covers complementary diagnostic
angles with strong debugging value and little
redundancy.

Example Worse Set:
- Q1 asks where the bug is implemented.
- Q2 asks which file contains the bug logic.
- Why worse: repeated intent, weak
complementarity, and poor effective coverage.

## Output Format

Return only valid JSON.

{
  "scores": {
    "coverage": 1
  },
  "summary": "One concise sentence explaining
  the judgment.",
  "dimension_rationale": {
    "coverage": "Short explanation referencing
     relevance, complementarity, redundancy,
    and debugging usefulness."
  }
}
```

3) Voting:

System Prompt

```
You are a rigorous evaluator comparing two
candidate sets of repository-focused
diagnostic questions generated for the same
GitHub issue.
```

```
Evaluate the two sets only using the provided
problem statement and the question texts.

Your goal is to decide which set is overall
more useful for helping a developer understand
, localize, and resolve the issue.

Bias controls:
- Ignore presentation order. Evaluate only by
merit.
- Do not reward verbosity, larger set size, or
 more complex language by default.
- Prefer concise, high-signal, repository-
grounded questions over bloated or generic
ones.
- If the two sets are functionally equivalent
for debugging, return TIE instead of forcing a
 winner.
```

User Prompt

```
## Input

Problem Statement:
{problem_statement}

Set A Questions:
{questions_a}

Set B Questions:
{questions_b}

## Task

Compare Set A and Set B holistically and
choose the better set.

## Decision Criteria

Compare the two sets across these dimensions:

1. Relevance
- Which set better targets the core failure
mechanism or likely repair path?

2. Repository Answerability
- Which set is more clearly answerable from
repository code, tests, config, or docs rather
 than generic software reasoning or outside
knowledge?

3. Diagnostic Utility
- Which set would do more to reduce debugging
uncertainty and narrow the search space?

4. Reasoning Depth
- Which set more often asks for meaningful
behavioral or structural reasoning rather than
 superficial lookup?

5. Coverage and Efficiency
- Which set better covers complementary
diagnostic angles while minimizing semantic
overlap and wasted question slots?

## Guidance

Repository Answerability is about where the
answer comes from.
Diagnostic Utility is about how much the
answer would help debugging.
Do not conflate them.

Do not reward complexity for its own sake.
A simpler but sharper set should beat a more
verbose but less useful set.

Do not reward a set just because it has more
questions.
Better coverage means stronger complementary
value, not larger quantity.

Return TIE when the practical debugging
benefit is effectively the same.

## Calibration Examples

Example Better Set:
- One question asks where the relevant
subsystem is implemented.
- One asks how the failing control flow or
state transition currently behaves.
- One asks about a boundary condition,
validation path, or missing test.
- Why better: strong complementarity, high
debugging value, low redundancy.

Example Worse Set:
- Multiple questions all ask where the bug is
located, which file is responsible, or where
the logic lives.
- Why worse: repetitive, weakly complementary,
 wastes question slots.

## Output Format

Return only valid JSON.

{
  "winner": "A",
  "confidence": 1,
  "summary": "One or two concise sentences
  explaining the decisive difference."
}

Notes:
- winner must be one of: A, B, TIE
- confidence must be an integer from 1 to 10
```

II. Supplementary Experimental Analysis

A. Stage-wise Cost and Time Analysis

Table I in the main paper reports the aggregate average time per instance. Table I provides a finer-grained breakdown, separating the pre-repair exploration stage from the downstream repair stage for both wall-clock time and monetary cost.

TABLE I
STAGE-WISE TIME AND COST PER INSTANCE.

		Time (s)			Cost ($)		
Method	**Model**	**Pre**	**Repair**	**Total**	**Pre**	**Repair**	**Total**
Mini-SWE-Agent	GPT-5-mini	0	187	187	0.000	0.024	0.024
	DeepSeek-V3.2	0	815	815	0.000	0.055	0.055
LocAgent	GPT-5-mini	106	331	437	0.023	0.025	0.048
	DeepSeek-V3.2	167	879	1046	0.021	0.038	0.060
CoSIL	GPT-5-mini	42	182	224	0.007	0.028	0.035
	DeepSeek-V3.2	31	719	750	0.004	0.055	0.059
LingmaAgent	GPT-5-mini	651	171	823	0.286	0.022	0.309
	DeepSeek-V3.2	640	781	1421	0.106	0.049	0.155
SWE-Debate	GPT-5-mini	1392	160	1552	0.714	0.024	0.738
	DeepSeek-V3.2	1675	843	2517	0.329	0.053	0.382
ACQUIRE	GPT-5-mini	125	177	302	0.027	0.026	0.054
	DeepSeek-V3.2	259	783	1042	0.022	0.051	0.073

Note. Results are measured on SWE-bench Verified (500 instances). "Pre" denotes the pre-repair stage; "Repair" denotes the downstream repair agent.

The stage-wise breakdown reveals several findings. First, compared with LingmaAgent and SWE-Debate, ACQUIRE's pre-repair stage is 2.5–11× faster and 5–26× cheaper, while achieving the best Pass@1. Second, compared with LocAgent and CoSIL, ACQUIRE's pre-repair overhead is moderate, and total cost stays comparable, while delivering a clearly higher Pass@1. Third, ACQUIRE adds only modest end-to-end overhead over the bare Mini-SWE-Agent (∼115 s / ∼227 s and $0.030 / $0.018 per instance) for the +3.8 / +4.4 Pass@1 gain. In addition, since the N Answerer instances are independent, the Answerer stage can be parallelized, which would further compress wall-clock time without changing total computation.

B. Agent Round and API-call Analysis

1) Agent Round Distribution: Figure 1 plots the distribution of agent rounds with and without QA injection under DeepSeek-V3.2. On the full set of 500 instances (left), QA injection shifts the distribution leftward, reducing the mean number of rounds by 7.1%. The shift becomes far more pronounced on the 44 Fail→Pass instances (right), where the mean reduction reaches 17.1%. This confirms that QA injection directly mitigates the disproportionate cost of knowledge-deficient repair attempts [8], [10], by front-loading the missing repository understanding and bypassing expensive trial-and-error loops.

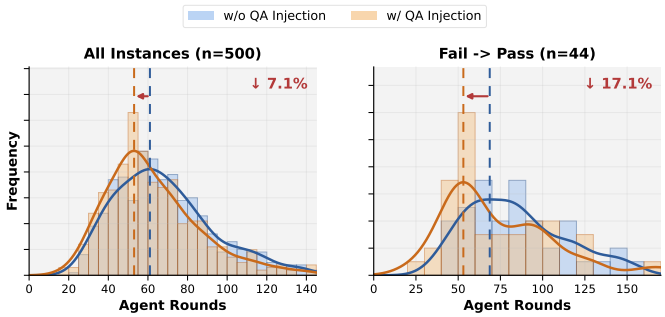

Fig. 1. Agent round distribution w/o vs. w/ QA injection under DeepSeek-V3.2.

2) API-call Shift by Transition Type: To further examine how QA injection affects exploration effort across different outcome groups, we compute the signed difference in the number of API calls between ACQUIRE and Mini-SWE-Agent ($\Delta = \text{API}_{ACQUIRE} - \text{API}_{\text{Mini-SWE-Agent}}$) for each instance and report the results grouped by transition type in Table II.

TABLE II
API-CALL SHIFT BETWEEN MINI-SWE-AGENT AND ACQUIRE ON SWE-BENCH VERIFIED UNDER DEEPSEEK-V3.2.

Group	**#Cases**	**Decrease**	**Same**	**Increase**	**Mean Δ**
Fail→Pass	44	32	0	12	−15.18
Pass→Fail overall	22	10	0	12	+0.55
Misleading	5	1	0	4	+11.40
Non-misleading	17	9	0	8	−3.64

Fail→Pass cases show a large reduction in API calls (mean $\Delta = -15.18$), consistent with QA reducing trial-and-error exploration. The aggregate Pass→Fail increase is small (+0.55) but is driven almost entirely by the misleading subset (+11.40); non-misleading Pass→Fail cases actually show a decrease (−3.64) on average. This confirms that QA shortens successful repairs, while misleading QA can increase exploration effort before failure.

C. Characterization of Generated QA

Table III summarizes the knowledge carried by the generated QA pairs across four categories. The distribution is heavily skewed toward *Mechanism & Behavior* (74.4%), reflecting that most issues demand understanding of internal logic flows, data transformations, and exception paths before a correct fix can be crafted. The remaining quarter is shared among *Design & Usage* (13.9%), *Locating & Structure* (9.5%), and *Ecosystem & Standards* (2.2%), which together supply complementary knowledge dimensions such as API contracts, dependency anchoring, and external constraints.

Although the 44 Fail→Pass instances follow a similar Mechanism-dominant distribution, 20 of the 44 instances (45.5%) require at least one question from a non-Mechanism category: *Design & Usage* in 11 cases, *Locating & Structure* in 7, and *Ecosystem & Standards* in 2. This suggests that for a

TABLE III
CHARACTERIZATION OF GENERATED QA PAIRS ACROSS KNOWLEDGE CATEGORIES.

Category	Prop.	Prominent Knowledge	Typical Answer Signals	Repair Contribution
Mechanism & Behavior	74.4%	Logic flow, data transform, error & exception	Explanation, code snippet, line reference, error & trace	Finds causal fix points and clarifies behavior boundaries.
Design & Usage	13.9%	API contract, design intent	Code entity, usage example, caveat	Constrains valid edits and preserves architectural consistency.
Locating & Structure	9.5%	Implementation site, definition site	File path, code entity, line reference	Narrows search space and anchors dependencies.
Ecosystem & Standards	2.2%	Protocol standard, library behavior	Caveat, usage example, summary	Prevents external inconsistency and regressions.

substantial portion of previously failed instances, Mechanism knowledge alone was insufficient, and cross-category combinations contributed to the successful fix, consistent with the coverage advantage reported in the category-guided ablation (Section 5.3 of the main paper).

III. HUMAN AUDIT AND REGRESSION DETAILS

This section provides supplementary details for the human audit and regression analysis reported in Section 5.2 of the main paper.

A. Minor Deviation Breakdown

Among the 132 QA pairs labeled *Supported with minor deviations*, we categorize the imperfections into three broad types:

- **Localized reference or detail inaccuracies (67.9%).** The main explanation is supported, but some local details such as code references, examples, function ownership, API details, or line ranges are imprecise.
- **Evidence-scope overreach (24.6%).** The answer extrapolates beyond directly verifiable repository evidence, e.g., by stating intent, broad causal explanations, external ecosystem behavior, or known limitations.
- **Implementation boundary overgeneralization (7.5%).** The described mechanism is broadly correct, but the answer overgeneralizes special cases, branch-specific behavior, or boundary conditions.

B. Cases Containing Ungrounded Claims

The two QA pairs labeled *Contains ungrounded claim* still contained useful repository-grounded information, but their central mechanism claims were not supported by the code.

- `django__django-12663_q0`. The answer correctly identified the relevant lookup-preparation path and field-preparation logic, but its central mechanism claim was wrong: it claimed that `SimpleLazyObject` inherits from `Promise` and would therefore be automatically evaluated by the `Promise` branch in `Field.get_prep_value()`. The repository instead defines `SimpleLazyObject` under the `LazyObject` hierarchy, not `Promise`, so this automatic `Promise`-based evaluation claim is contradicted by the code.
- `sympy__sympy-23534_q0`. The answer correctly located `symbols()` and described the normal use of `cls`, but incorrectly claimed that `cls` is propagated through nested tuple recursion; the implementation recursively calls `symbols(name, **args)` without forwarding `cls=cls`.

C. Pass-to-Fail Regression Details

We manually inspect all 22 Pass→Fail regression trajectories analyzed in Section 5.2 of the main paper to distinguish failures caused by misleading QA from failures where QA was not the primary cause of regression. We label a case as *misleading* when the dominant downstream effect of the injected QA on the Resolver's repair behavior is misleading. Under this criterion, 5 of the 22 Pass→Fail cases are misleading, while the remaining 17 are non-misleading.

1) Misleading Cases: We further categorize the 5 misleading Pass→Fail cases by the Resolver's primary repair behavior, as summarized in Table IV.

TABLE IV
REPAIR-BEHAVIOR BREAKDOWN OF THE 5 MISLEADING PASS→FAIL CASES UNDER DEEPSEEK-V3.2.

Repair behavior	#	Relation to QA
Wrong modification location	2	QA emphasizes a related but non-gold location, which the Resolver keeps searching and editing.
Wrong repair logic	2	QA-provided mechanisms or context are turned into logic inconsistent with the gold fix.
Overbroad edits	1	QA provides an overly broad set of related methods or files, leading to unnecessary edits.

In these cases, the QA often still contains repository-grounded facts, relevant code locations, or locally correct mechanism descriptions. The failure mode is that the QA provides a plausible but incorrect repair framing, and the Resolver repeatedly follows the QA-highlighted location, mechanism, or scope during search, editing, and testing, eventually deviating from the gold fix.

Cross-referencing with the answer-level audit (Section III-A), among the 10 QA pairs in the 5 misleading cases,

only 1 was labeled *Contains ungrounded claim*; the remaining 9 were *Supported*. The clearest ungrounded-claim example is `django__django-12663_q0` (see Section III-B), where the wrong `SimpleLazyObject`/`Promise` inheritance claim makes the failure look like a lazy-object value-preparation issue and steers the Resolver toward the wrong repair direction. In contrast, many other misleading cases are more subtle: the QA may describe relevant repository mechanisms correctly, but those mechanisms are not the right repair target.

2) Non-misleading Cases: For the remaining 17 non-misleading Pass→Fail cases, QA generally played a more constructive role: it often helped the Resolver identify relevant files, understand the failure mechanism, or make partial progress toward the gold fix. The eventual failures were more often due to Resolver-side behavior, such as overgeneralizing a correct clue, implementing only part of the required fix, adding extra non-gold behavior, or submitting a patch that did not preserve the QA-guided repair direction.

We note that the Resolver's prompt explicitly instructs it to treat injected QA as auxiliary context that must be cross-checked against the repository, and to discard or override QA whenever it conflicts with directly observed code, command outputs, or test feedback rather than blindly following it. Therefore, these cases should not be interpreted as QA misleading the Resolver; rather, they show limitations in how the Resolver uses otherwise helpful or partially helpful QA, even under a cautious-use prompting policy.

This regression analysis confirms a limitation of ACQUIRE: QA can influence repair behavior negatively in some cases. However, imperfect intermediate artifacts are an inherent limitation of all LLM-based methods, as pre-fix retrieval, plan-then-fix, agentic patch generation, and direct end-to-end repair are all subject to hallucinated or partially incorrect intermediate outputs, rather than a defect specific to QA injection. The audit and regression analysis quantify this risk concretely under ACQUIRE and show that misleading QA accounts for only 5 of the 22 inspected regressions, while the framework yields a net $+22$ instance gain, indicating that the risk is bounded and substantially outweighed by the benefit.