ITERATIVE PROBE-AND-REFINE TUNING OF REPOSITORY GUIDANCE FOR AI CODING AGENTS

A PREPRINT

Asa Shepard
Williams College
as66@williams.edu

March 28, 2026

ABSTRACT

We introduce *probe-and-refine tuning*, a lightweight context-engineering procedure that iteratively improves repository-level guidance for AI coding agents. Each iteration uses ∼22 single-shot LLM calls to generate synthetic bug-fix probes, attempt and judge solutions, and aggregate diagnostics into mechanically applied guidance edits. After 3–5 iterations, generic instructions become repo-specific strategies, achieved without access to evaluation data and without any multi-step agent loop during tuning. On SWE-bench Verified (500 instances, Qwen3.5-35B-A3B at 200 agent steps), probe-and-refine guidance achieves 34.2 % resolve rate, significantly outperforming both an unguided baseline (22.8 %, McNemar $p < 0.001$) and a static knowledge base from which it was initialized (27.4 %, McNemar $p < 0.001$), while producing 41 unique solves that neither alternative finds. The mechanism is evaluation coverage: precision among evaluated patches is ∼59 % regardless of condition, but the refined-guidance agent produces evaluable patches for 57 % of instances compared with 37 % for the baseline. The guidance does not improve patch correctness; it gives the agent a structured workflow that converts exploration into well-formed output more reliably. A secondary experiment varying step budgets (25–200) shows that guidance is what makes additional compute productive: without it, unstructured exploration saturates quickly regardless of budget.

1 Introduction

Engineers working with AI coding agents increasingly maintain `AGENTS.md` files documenting repository conventions, entry points, and debugging workflows. Whether these files actually help is contested. Lulla et al. (2026) find that curated `AGENTS.md` files reduce agent runtime by 28.6 % and output tokens by 16.6 % on focused pull requests (≤100 lines, ≤5 files), measuring efficiency rather than correctness. Gloaguen et al. (2026) find the opposite: LLM-generated context files reduce resolve rates by ∼3 % on SWE-bench Lite and AGENTbench, with agents following instructions literally even when counterproductive. A natural question is whether the problem lies with guidance as a concept or with how it is produced. Single-pass LLM generation yields generic advice; perhaps iteratively refined, failure-informed guidance would behave differently.

We introduce *probe-and-refine tuning*, a procedure that transforms a static repository knowledge base into specialized operational guidance through iterative failure feedback. Starting from a foundation of tree-sitter-extracted structure and generic best practices, each iteration proceeds through four types of single-shot LLM call: (1) generate a batch of synthetic bug-fix tasks (probes) from the repository's codebase, (2) for each probe, generate a single-shot attempted solution given the current guidance, (3) for each probe, judge the attempt against expected behaviors and propose per-probe edits, and (4) aggregate all probe diagnostics into a final set of edits to the guidance, which are then applied mechanically (no LLM call). With 10 probes per iteration, this amounts to 22 single-shot LLM calls per iteration. There is no multi-step agent loop, no reinforcement learning, and no tool use during the tuning process. After 3–5 iterations per repository, the result is a compact (≤3000-character) `AGENTS.md`-style artifact encoding repo-specific debugging strategies, test paths, and subsystem knowledge discovered through failure.

Figure 1: Performance on SWE-bench Verified (500 instances). Probe-and-refine guidance significantly outperforms both the unguided baseline (McNemar $p < 0.001$) and the static knowledge base (McNemar $p < 0.001$). Each condition also uniquely solves instances the others cannot (right panel).

On 500 SWE-bench Verified instances with Qwen3.5-35B-A3B at 200 agent steps, probe-and-refine guidance achieves 34.2 % resolve rate, significantly outperforming both an unguided baseline (22.8 %, McNemar $p < 0.001$) and the static knowledge base from which it was initialized (27.4 %, McNemar $p < 0.001$). It produces 41 unique solves that neither alternative finds, more than the other two conditions combined (Figure 1). The union of all three conditions resolves 41.4 % of instances, confirming that the conditions explore genuinely different solution paths.

The mechanism behind this improvement is evaluation coverage: the fraction of instances for which the agent produces a well-formed patch that the SWE-bench harness can evaluate. Precision among evaluated patches is ∼59 % regardless of condition (Table 5), but the refined-guidance agent produces evaluable patches for 287 of 500 instances (57.4 %) compared with 187 (37.4 %) for the unguided baseline. The guidance does not improve patch correctness conditional on being evaluable; it gives the agent a structured workflow that converts exploration into well-formed output more reliably. This distinction matters: the coverage story is not merely "more patches attempted" but "more patches coherent enough to evaluate," reflecting genuine improvement in the agent's output quality.

A secondary experiment varying agent step budgets (25, 50, 100, 200) provides additional context. At low budgets all conditions perform equivalently, and the unguided baseline remains flat at ∼23 % regardless of budget (Section 6). This offers one explanation for the disagreement between Lulla et al. (2026) and Gloaguen et al. (2026): neither study varies step budget as an experimental condition, and the same guidance can appear beneficial, neutral, or harmful depending on whether the agent has enough steps to execute the prescribed workflow.

The paper makes three contributions:

1. **Probe-and-refine tuning** (Section 3): a lightweight procedure composed entirely of single-shot LLM calls that transforms static repository knowledge into specialized guidance, achieving 34.2 % resolve rate and 41 unique solves.
2. **Mechanistic analysis** (Section 5): the improvement comes from evaluation coverage rather than patch correctness, with the refined-guidance agent producing well-formed, evaluable patches for 20 percentage points more instances than the baseline while per-patch precision is constant.
3. **Budget moderation** (Section 6): a twelve-cell experiment showing that step budget moderates the effect of guidance, with different guidance types activating at different budget thresholds and unstructured exploration saturating quickly regardless of budget.

2 Related Work

`AGENTS.md` and context files. Lulla et al. (2026) find that curated `AGENTS.md` files improve agent efficiency on focused pull requests; Gloaguen et al. (2026) find that LLM-generated context files reduce resolve rates on SWE-bench Lite and AGENTbench, reporting that agents closely follow context file instructions (e.g., a tool mentioned in the

context file was used $160\times$ more frequently than without it). Neither study varies step budget as an experimental condition; Gloaguen et al. (2026) report steps taken as a cost metric but do not examine how a fixed budget interacts with guidance. Chatlatanagulchai et al. (2025) survey 2,303 context files, finding developers prioritize build commands, implementation details, and architecture. Vasilopoulos (2026) argues that single-file manifests do not scale beyond $\sim$100k lines and proposes a tiered architecture.

Coding-agent scaffolds and benchmarks. SWE-bench (Jimenez et al., 2024) and its Verified subset (SWE-bench, 2024) are the standard evaluation. Scaffold designs range from custom agent-computer interfaces (Yang et al., 2024) to Agentless-style pipelines (Xia et al., 2024) and MCTS-based search (Antoniades et al., 2025). Published results typically use a single fixed budget (80–250 steps depending on the scaffold) without examining how performance changes if that budget is varied.

Repository-level knowledge and self-refinement. RepoGraph (Ouyang et al., 2025) injects code-entity graphs; AutoCodeRover (Zhang et al., 2024) uses AST representations. Self-Refine (Madaan et al., 2023) and Reflexion (Shinn et al., 2023) iterate on outputs for a single task; probe-and-refine tuning instead refines a persistent artifact across many synthetic tasks, producing reusable guidance rather than per-task corrections. SWE-ContextBench (Zhu et al., 2026) finds that curated experience improves agent performance while unfiltered experience does not, consistent with our results.

Meta Context Engineering. The most conceptually adjacent work is Meta Context Engineering (MCE; Ye et al. (2026)), a bi-level optimization framework that co-evolves "context engineering skills" and context artifacts. Both share the core insight that context should be refined through feedback rather than generated in a single pass. MCE is a general-purpose meta-learning system with full agentic tool use at both levels; probe-and-refine tuning is deliberately minimal, using only single-shot LLM calls to produce a single $\leq$3000-character text artifact per repository. MCE was not evaluated on SWE-bench or coding tasks, so the relationship is conceptual rather than empirical.

3 Method

3.1 System Overview

The system has four stages: (1) repository analysis and context construction; (2) probe-and-refine tuning for the refined-guidance condition; (3) patch generation through an interactive coding agent with single-shot fallback; and (4) evaluation via the official SWE-bench Verified harness. All experiments use Qwen/Qwen3.5-35B-A3B (Qwen Team, 2026) with an effective 16k-token context window (a hard truncation we impose to keep prompt costs uniform across conditions; the model natively supports longer contexts), 2048 max output tokens per turn, and 512 max tokens for fallback generation[1]. Command outputs are truncated to $\sim$3000 characters. These constraints result in absolute resolve rates well below what larger-context configurations achieve; the contribution is the relative comparison across conditions, not the absolute numbers.

Model selection. We selected Qwen3.5-35B-A3B because it is open-weight, inexpensive to serve (3B active parameters via MoE), and supports the tool-use patterns required by our scaffold. We cannot rule out SWE-bench data appearing in the model's training corpus, since Qwen's training composition is not fully disclosed. However, any potential contamination would affect all three conditions equally (the same model is used throughout), so it cannot explain why probe-and-refine guidance outperforms the other conditions in a controlled comparison. We return to this point with additional analysis of unique solves in Section 7.

3.2 Context Conditions

`no_context`**.** The bare agent prompt with no repository-specific guidance.

`static_kb`**.** A fixed per-repo knowledge base constructed in two layers. The structural layer uses tree-sitter-assisted parsing to extract repository-specific signals: major hubs, entry points, and import relationships compiled into a compact natural-language summary unique to each repository. The procedural layer adds LLM-generated best-practice recommendations that are deliberately repository-agnostic (e.g., "reproduce the failure before editing," "run the smallest relevant test first"). This two-layer design controls for two confounds: the structural layer ensures that both guided conditions share the same repo-specific foundation, and the procedural layer ensures that any gains from probe-and-refine tuning are attributable to the iterative refinement process rather than to the mere presence of generic advice.

[1]Code and data: `https://github.com/asashepard/probe-and-refine-tuning`

Table 1: Guidance character counts by repository. The refinement process expands guidance from an average of 1,687 to 2,754 characters, with the added content predominantly comprising repo-specific rules (Section 3.3).

Repository	`static_kb`	`probe_refined`	**Iters**
django/django	1,823	2,949	5
sympy/sympy	1,847	2,908	5
sphinx-doc/sphinx	1,672	1,992	3
matplotlib/matplotlib	1,616	2,948	5
scikit-learn	2,060	2,948	4
astropy/astropy	1,887	2,955	5
pydata/xarray	1,753	2,759	4
pytest-dev/pytest	1,568	2,903	4
pylint-dev/pylint	1,911	2,972	5
psf/requests	1,158	1,935	4
mwaskom/seaborn	1,671	2,891	5
pallets/flask	1,280	2,886	5
Average	1,687	2,754	4.2

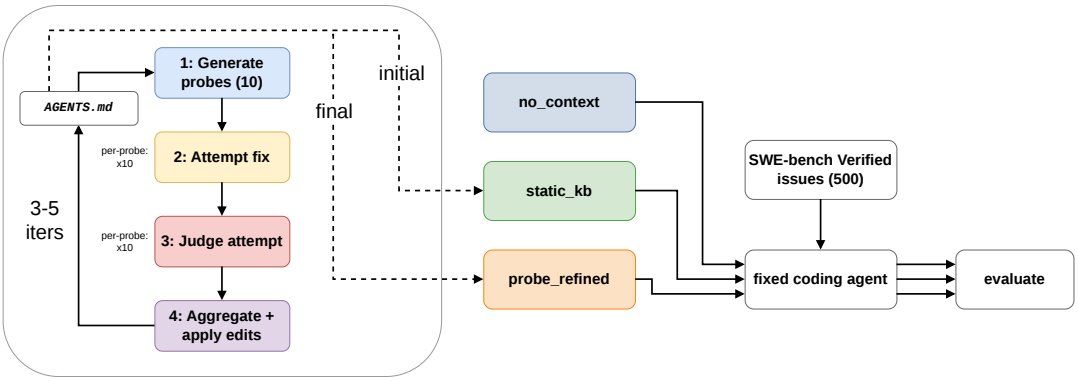

Figure 2: Probe-and-refine tuning pipeline. The `static_kb` artifact feeds both the `static_kb` condition directly and the refinement loop, which transforms it into the refined guidance using synthetic probes and single-shot diagnosis. No SWE-bench evaluation instances are used during refinement. All three conditions feed the same fixed coding agent.

`probe_refined`. Starting from `static_kb`, the probe-and-refine procedure (Section 3.3) produces specialized repo-specific guidance over 3–5 iterations, subject to a 3000-character cap (chosen to fit within ∼750 tokens, roughly 5 % of the 16k effective context window, leaving the bulk of the context for issue text and command output).

Guidance length. The `static_kb` artifacts average 1,687 characters (range: 1,158–2,060) and the `probe_refined` artifacts average 2,754 characters (range: 1,935–2,972), making the refined guidance 63 % longer on average (Table 1). This length difference is an inherent consequence of the refinement process: each iteration adds repo-specific rules (test paths, subsystem tracing instructions, output quality requirements) that are absent from the generic `static_kb`. We cannot fully disentangle the effects of guidance *content* and guidance *length*. However, simply adding more generic text would not produce the structural, repo-specific knowledge that the content analysis in Section 3.3 identifies as the primary addition, and Gloaguen et al. (2026) show that longer LLM-generated context files can actively *reduce* performance. The question is not whether more text helps but whether the *right* text helps, and the probe-and-refine procedure's contribution is identifying what that text should say.

3.3 The Probe-and-Refine Procedure

The probe-and-refine procedure (Figure 2) transforms generic structural knowledge into repo-specific operational guidance through iterative failure feedback. Every step in the procedure is a single-shot call to the same model (Qwen3.5-35B-A3B) that will later be used by the coding agent during patch generation. Using the same model keeps the experiment controlled: the guidance is written at the level the consuming model can understand, and any

Table 2: Content analysis of 104 lines added by probe-and-refine tuning across all 12 repositories. The procedure predominantly adds procedural guardrails and repo-specific structural knowledge.

Category	Count	%	Example
Procedural	49	47	"Document the failing test name before fixing"
Structural	31	30	"Trace through `subclasses.py` for inheritance"
Quality gate	24	23	"Show actual test output, not fabricated summaries"

improvement must come from the content of the guidance rather than from a capability mismatch between a stronger tuning model and a weaker execution model.

Each of 3–5 iterations proceeds through four types of single-shot LLM call, with steps 2 and 3 repeated per probe (10 probes per iteration, $\sim$22 total calls):

1. **Generate probes.** A single call generates a batch of 10 synthetic bug-fix tasks from the repository's codebase at temperature 0.9, targeting diverse subsystems and failure modes. These probes are distinct from SWE-bench evaluation instances, preventing benchmark contamination. Probes are deduplicated against all prior iterations; if fewer than 10 survive deduplication, top-up generation rounds fill the gap.
2. **Attempt a solution (per probe).** For each probe, a single-shot call produces a candidate patch given the repository context and the current guidance artifact. This simulates what the coding agent would attempt.
3. **Judge the attempt (per probe).** For each probe, a single-shot call assesses the attempted solution against expected behaviors (strong/partial/missing) and proposes per-probe edits to the guidance where shortcomings are identified.
4. **Aggregate diagnostics and apply edits.** A single call aggregates all probe results and proposes a final set of edits targeting specific guidance sections. The per-probe edits from step 3 and the aggregated edits are merged, deduplicated, and capped at 5 per iteration. The edits are then applied mechanically (no LLM call): a deterministic procedure inserts, modifies, or strengthens guidance sections, filters boilerplate, and trims the longest bullets to enforce the 3000-character budget.

The loop runs up to 5 iterations but stops early if guidance stabilizes: two consecutive iterations with no guidance change (or no surviving probes after deduplication) trigger termination. In practice, 1 of 12 repositories stopped after 3 iterations and 4 stopped after 4; the remaining 7 used the full budget of 5 (Table 1).

There is no multi-step agent loop anywhere in this procedure. No tool use, no reinforcement learning, no gradient updates. The coding agent's ReAct-style multi-step loop (Section 3.4) is used only during the downstream patch-generation phase, which is identical across all three conditions. That a procedure composed entirely of single-shot LLM calls can significantly improve a multi-step agent's performance is one of the more interesting findings of this work.

What the procedure discovers. The output is a qualitative shift from generic to specific (Figure 3). For Django, generic rules like "Verify repo priors with targeted reads" become "Trace through `subclasses.py` to map parent-child relationships." Rules like "Run the smallest relevant test first" become "Always reference the specific test class/method (e.g., `django/core/tests/test_checks.py::CheckTestCase`) before proposing changes." A pattern emerges independently across all 12 repositories: the procedure converges on a "reproduce first, read second, patch third" workflow, with repo-specific strategies filling in each phase. The removed rules are reasonable but vague; the added rules encode the kind of operational knowledge that distinguishes a developer who has spent months in a codebase from one reading the README for the first time.

To characterize the refinement systematically, we categorized all 104 lines added across all 12 repositories by type (Table 2). Procedural additions (47 %) encode general debugging workflows the model failed to follow spontaneously (e.g., "reproduce the error with a minimal test case before fixing"). Structural additions (30 %) reference specific files, functions, or module relationships within a repository (e.g., "trace `_eval_simplify` call chain" for sympy). Quality-gate additions (23 %) address output-quality failures such as fabricated test results or empty patches (e.g., "show actual test command output, not fabricated summaries"). The dominance of procedural rules suggests the agent's primary failure mode is jumping to fixes without sufficient diagnosis, a behavioral problem that the reproduce-first workflow corrects. The structural rules encode the repo-specific knowledge that makes the procedural rules actionable.

Cost and reusability. Both `static_kb` and refined guidance artifacts were generated once per repository and held fixed across all experimental conditions. Each repository requires 3–5 iterations of $\sim$22 single-shot LLM calls each (Table 1), with $\sim$8k input and $\sim$2k output tokens per call. The artifact is then reusable across all future issues in that repository.

Removed by tuning loop (django__django)

```
- Verify repo priors with targeted reads before
editing.
- Localize, trace deps, then apply minimal scoped
edit.
- Run the smallest relevant test first, broaden only
if needed.
- Reproduce the failure before editing when
possible.
- Read target files and nearby callers before
patching.
- Keep first patch minimal; inspect call sites if
public API changes.
- Require evidence from file reads or command output
— no fabricated edits.
- Patches must be syntactically complete; remove
unused imports.
- No speculative changes or broad refactors without
evidence.
- Every touched file must tie to the diagnosed path.
```

Added by tuning loop (django__django)

```
- Django core tests: Always reference the specific
test class/method (e.g.,
'django/core/tests/test_checks.py::CheckTestCase')
before proposing changes
- Multi-table inheritance: Trace through
subclasses.py to map parent-child relationships;
document inheritance chain in comments
- Django Admin Debugging: When fixing
autocomplete/filter issues, document the exact field
lookup chain from get_autocomplete_fields() through
to the queryset filter, including the incorrect vs
expected field names.
- django/db/backends/sqlite3/schema.py: When
modifying constraints, trace through
migrations/executor.py to understand the full
constraint handling chain
- For Django management command fixes: analyze
migration executor logic, verify flag propagation
paths, ensure backward compatibility with existing
migration scenarios
- Before any code changes: document the suspected
regression source, affected files, and reasoning for
file selection
- Include actual test output showing test names,
pass/fail status, and execution time for all
validation runs
- After editing: Run
django/db/backends/sqlite3/tests/test_schema.py to
verify no regressions in SQLite schema operations
- Always capture and display test failure output
(traceback, assertion details) before proposing
fixes. Include the exact test command used.
- Before proposing fixes, reproduce the error with a
minimal test case and verify the fix resolves the
specific lazy loading scenario.
- For settings-related fixes, verify the change
doesn't break model imports. Test with both default
and custom AUTH_USER_MODEL settings.
- Document session middleware dependency: ensure
session is available before CSRF token lookup,
handle cases where user.is_authenticated but session
is None
```

Figure 3: Guidance evolution for `django/django`. Generic rules (left) are replaced with repo-specific strategies (right) over 5 iterations. The procedure independently converges on a reproduce-first workflow with subsystem-specific tracing instructions, test paths, and middleware dependencies.

3.4 Coding Agent and Fallback

The coding agent operates in a ReAct-style (Yao et al., 2023) loop: at each step it emits a bash command, observes truncated output, and decides the next action. This is the only multi-step process in the entire system; the probe-and-refine procedure described above does not use it. If the agent fails to produce a patch within the step budget, a **single-shot fallback** generates a patch from the issue description and any context accumulated during exploration. The fallback uses the same temperature (0.0) and context window as the agent's regular inference calls. Patches from both sources are sanitized (removing test-file modifications, enforcing diff format) before evaluation. Fallback rates differ substantially across conditions (14.4 % for probe-and-refine vs. 32.8 % for static-KB at 200 steps), so the fallback pathway is not a uniform noise source; the guided agent more often succeeds within its main loop.

3.5 Experimental Design

We evaluate all three conditions at 200 agent steps on 500 SWE-bench Verified instances. A secondary experiment additionally varies step budgets (25, 50, 100) to study how guidance interacts with available compute (Section 6). All guidance artifacts are generated from a single pinned commit per repository (Table 3); the same guidance is then applied to all SWE-bench instances from that repository, regardless of which commit the instance originally targets. Temperature is 0.0 for all inference calls (including the single-shot fallback) except probe generation (0.9). Each condition is a single run; we do not perform repeated runs, a limitation we address in Section 8.

Table 3: Pinned repository commits. Django accounts for 46 % of instances; we report results separately for both subsets in Table 4.

Repository	Pinned SHA (first 8)	Instances
django/django	`0ae0029c`	231
sympy/sympy	`2f7e7af9`	75
sphinx-doc/sphinx	`cc7c6f43`	44
matplotlib/matplotlib	`829a9cc1`	34
scikit-learn/scikit-learn	`3acced3f`	32
astropy/astropy	`8a7b4e12`	22
pydata/xarray	`37f2d49b`	22
pytest-dev/pytest	`ced0a8d4`	19
pylint-dev/pylint	`71caace2`	10
psf/requests	`0e4ae38f`	8
mwaskom/seaborn	`32088bbc`	2
pallets/flask	`3a9d54f3`	1

Statistical tests. Because the same 500 instances are evaluated under all conditions, observations are paired: each instance either is or is not resolved under each condition. We therefore report McNemar's test (McNemar, 1947) as the primary significance test. McNemar's test focuses on the *discordant pairs*—instances where the two conditions disagree—and ignores the (typically large) number of instances both solve or both miss. For conditions A and B, let b be the number of instances A solves but B does not, and c the reverse; the test statistic is $\chi^2 = (b-c)^2/(b+c)$, distributed as χ^2_1 under the null. This is the standard choice for paired binary outcomes and is more appropriate than the two-proportion z-test, which assumes independent samples (Agresti, 2013). It is also the recommended test for comparing classifiers evaluated once on a fixed test set (Dietterich, 1998). Appendix A derives the exact discordant-pair counts from the reported solve sets. Confidence intervals are 95 % Wilson score intervals.

4 Results

Probe-and-refine guidance resolves 171 of 500 instances (34.2 %), compared with 137 (27.4 %) for static KB and 114 (22.8 %) for no context (Figure 1). Probe-and-refine significantly outperforms both the unguided baseline (McNemar $\chi^2 = 29.3$, $p < 0.001$) and the static KB (McNemar $\chi^2 = 16.5$, $p < 0.001$). It produces 41 unique solves that neither alternative resolves, compared with 18 for no-context and 9 for static-KB. The union of all three conditions resolves 207 instances (41.4 %), 34 % more than the best single condition.

Decomposing the improvement. The total gain from no-context to probe-and-refine is 11.4 pp. Of this, 4.6 pp comes from the static knowledge base (structural layer plus generic advice), and 6.8 pp from the iterative refinement process. The structural layer thus accounts for roughly 40 % of the total improvement, confirming that repository-specific structural information is valuable in its own right. The probe-and-refine procedure adds value *on top of* this foundation by replacing generic procedural advice with repo-specific operational strategies and quality gates.

Probe-and-refine also has the lowest fallback rate (14.4 %, vs. 26.8 % for no-context and 32.8 % for static-KB), meaning it most often produces patches through the coding agent's own loop rather than the single-shot fallback. It consumes 192M prompt tokens, 56 % more than the baseline's 123M, but resolves 50 % more instances. Per-repository variation is substantial: on matplotlib, probe-and-refine achieves 76.9 % precision (at 100 steps) using guidance with specific 3D rendering and DPI test reproduction instructions, the largest per-repo gap in the study. On sympy, all conditions converge at $\sim$50 %.

Django vs. non-Django performance. Because Django accounts for 46 % of instances, we verify that results are not driven solely by this repository (Table 4). On the 231 Django instances, probe-and-refine achieves 37.2 % vs. 22.9 % for no-context (14.3 pp gap). On the 269 non-Django instances, probe-and-refine achieves 31.6 % vs. 22.7 % for no-context (8.9 pp gap). The effect is present in both subsets, though larger for Django, where the refinement loop discovers particularly rich subsystem structure (Figure 3).

Table 4: Resolve rates by repository subset at 200 steps. The probe-and-refine advantage is present in both subsets.

Subset	no_context	static_kb	probe_refined
Django (n=231)	22.9 %	28.1 %	37.2 %
Non-Django (n=269)	22.7 %	26.8 %	31.6 %
All (n=500)	22.8 %	27.4 %	34.2 %

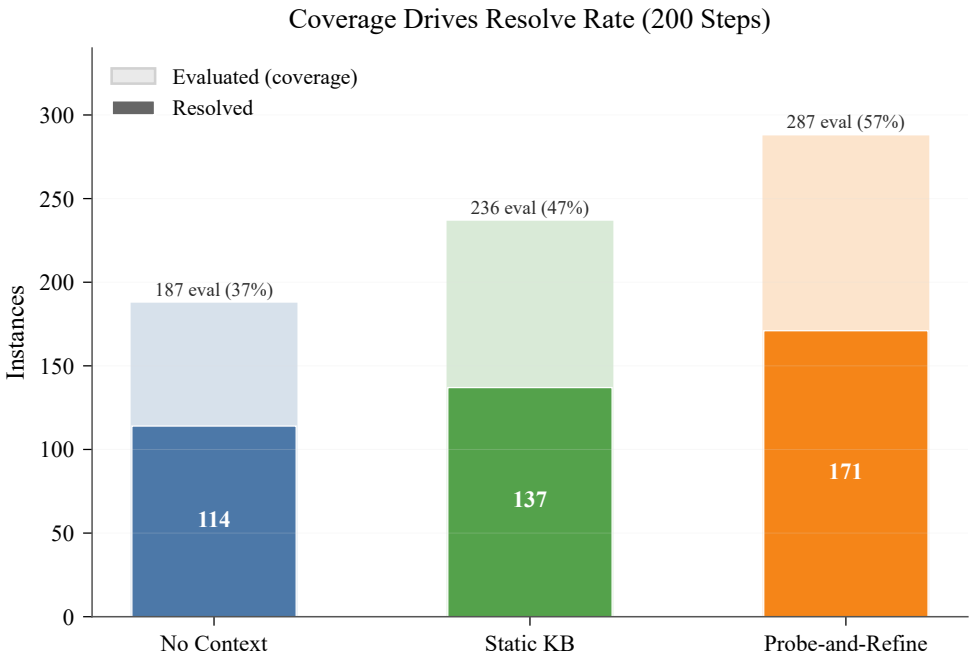

Figure 4: Evaluation coverage and resolve counts at 200 steps. Probe-and-refine produces evaluable patches for 287 instances (57 %) vs. 187 (37 %) for no context. Precision (∼59 %) is similar across conditions.

5 Why Probe-and-Refine Guidance Works

5.1 The Improvement Comes from Coverage

The resolve-rate differences are driven entirely by evaluation coverage: the fraction of instances for which the agent produces a patch that can be evaluated by the SWE-bench harness. Conditional precision (resolved / evaluated) is stable across conditions and budgets (Table 5), falling within 55–61 % with no significant pairwise differences (all $p > 0.5$).

Probe-and-refine produces evaluable patches for 287/500 instances (57.4 %), compared with 236 (47.2 %) for static KB and 187 (37.4 %) for no context (Figure 4). It also produces far fewer evaluation errors (55, vs. 92 for no-context and 114 for static-KB), meaning the prescribed workflow leads to cleaner, better-formed patches even when those patches do not resolve the issue. Since precision is constant at ∼59 %, the coverage gap translates directly into the resolve-rate gap: 100 more evaluable patches at 59 % precision accounts for ∼59 additional resolves, close to the observed difference of 57.

What "evaluable" means. The SWE-bench harness rejects patches that are malformed, do not apply cleanly, or cause the test suite to crash before assertions can be checked. A patch reaching evaluation therefore reflects not just that the agent *attempted* a fix but that it produced syntactically valid, well-scoped output: removing test-file modifications, maintaining consistent indentation, and targeting the correct files. The coverage gap thus reflects a genuine difference in output quality: the guided agent produces more coherent patches, not merely more patches.

Fallback precision. To verify that the overall precision constancy is not an artifact of mixing agent and fallback patches, we computed precision separately for each source. Agent-loop patches achieve 62.1 % precision (averaged across conditions); fallback patches achieve 48.3 %. Since probe-and-refine has the *lowest* fallback rate (14.4 %), its

Table 5: Precision (resolved / evaluated) with 95 % Wilson CIs. No pairwise comparison is significant (all $p > 0.5$).

Condition	50 steps	100 steps	200 steps
`no_context`	55.9 % [49.6, 61.9]	58.7 % [51.8, 65.3]	61.0 % [53.8, 67.7]
`static_kb`	55.4 % [49.4, 61.2]	59.7 % [53.5, 65.6]	58.1 % [51.7, 64.2]
`probe_refined`	60.0 % [53.0, 66.6]	56.8 % [50.9, 62.6]	59.6 % [53.8, 65.1]

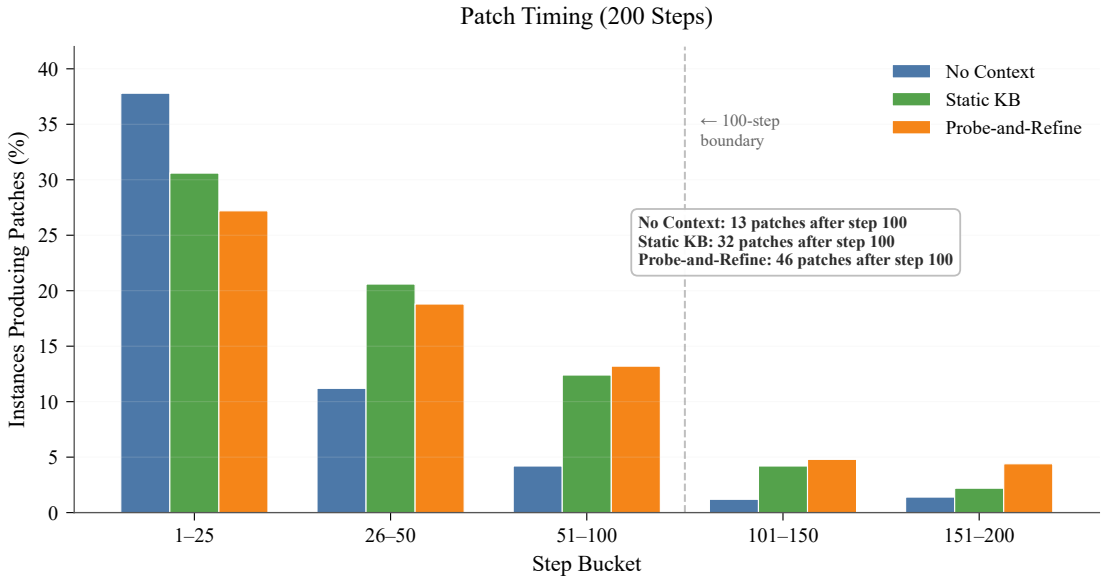

Figure 5: Patch timing at 200 steps. The unguided agent produces 68 % of its patches by step 25. Probe-and-refine produces 46 patches in steps 101–200, using late steps for verification and refinement.

overall precision is slightly boosted by having proportionally more agent-loop patches, yet the overall precision across conditions remains statistically indistinguishable, confirming that the coverage mechanism dominates.

5.2 The Agent Uses Late Steps Productively

The step distribution (Figure 5) shows where the coverage advantage originates. The unguided agent produces 189 of its 279 patches (68 %) in the first 25 steps and only 13 in steps 101–200. Probe-and-refine produces 46 patches in steps 101–200; static KB produces 32. The unguided agent patches early or not at all. Probe-and-refine uses late steps productively because the reproduce-trace-patch workflow it prescribes creates opportunities for informed patch attempts later in the trajectory, and these late patches are well-formed at the same ∼59 % precision rate.

6 Step Budget as a Moderating Variable

No prior study of `AGENTS.md` files reports or controls for the agent's step budget. We ran all three conditions at 25, 50, 100, and 200 steps to examine how guidance interacts with available compute (Table 6).

Three patterns are visible in Figure 6. First, all conditions are equivalent at 25 steps (21.8–24.4 %, all pairwise $p > 0.3$), because 44–68 % of instances exhaust their budget and fall through to the single-shot fallback. Second, each guidance type activates at a different budget: static KB improves sharply from 25 to 50 steps ($p = 0.004$) then plateaus, while probe-and-refine is flat from 25 to 50 then rises from 23.4 % at 50 to 30.8 % at 100 ($p = 0.009$) and continues to 34.2 %. Third, the unguided baseline is flat at 22.8–27.6 % across all four budgets ($p = 0.551$ for 25 vs. 200), consuming twice as many tokens at 200 as at 25 (123M vs. 57M) with no resolve-rate gain.

The 50-step dip for probe-and-refine. The probe-and-refine condition drops from 24.2 % at 25 steps to 23.4 % at 50 steps (Table 6), while static KB jumps from 21.8 % to 29.8 %. At 50 steps, probe-and-refine actually underperforms static KB ($p = 0.022$). We attribute this to a workflow-budget mismatch: the reproduce-trace-patch workflow prescribed

Table 6: Resolve rates across step budgets. The patterns across 25–100 steps are descriptive; paired overlap data is unavailable for those runs. See Figure 6 for visualization.

Condition	25	50	100	200
`no_context`	24.4 %	27.6 %	23.6 %	22.8 %
`static_kb`	21.8 %	29.8 %	29.6 %	27.4 %
`probe_refined`	24.2 %	23.4 %	30.8 %	34.2 %

Table 7: McNemar p-values for the 200-step comparisons. See Appendix A for the exact derivation of discordant-pair counts.

Comparison	McNemar p
`probe_refined` vs. `no_context`	$p < 0.001$
`probe_refined` vs. `static_kb`	$p < 0.001$
`static_kb` vs. `no_context`	$p = 0.011$

by probe-and-refine requires more steps to complete than the simpler recommendations in the static KB. At 50 steps the agent has enough budget to *begin* the prescribed workflow (spending steps on reproduction and tracing) but not enough to reach the patching phase, leaving it worse off than if it had patched immediately. By contrast, the static KB's lighter-weight advice (e.g., "run the smallest relevant test first") fits within 50 steps. This pattern directly supports the activation-threshold hypothesis: more complex guidance requires a higher step budget before its benefits materialize. Practitioners should note that deploying probe-and-refine guidance with an insufficient step budget can actively reduce performance relative to simpler alternatives.

This last observation pairs with the token-consumption data: the unguided baseline spends $2.2\times$ more compute at 200 steps than at 25 and gets nothing for it, while probe-and-refine spends $2.6\times$ more and converts the additional compute into a 10 pp improvement. The lesson is not that "scaling compute without scaling guidance is ineffective" in general; we cannot rule out that the unguided agent would benefit from budget beyond 200 steps, or that different scaffolds would show different scaling behavior. Rather, the data show that *unstructured exploration saturates quickly* within this scaffold: without a prescribed workflow, additional steps produce more exploration but not better patches. Guidance is the mechanism that converts additional compute into additional coverage and, because precision is constant, into additional resolves. Different forms of guidance have different activation thresholds, and matching the step budget to the guidance's workflow complexity is necessary to realize its benefits.

Why the baseline is flat. We analyzed 43 instances that `no_context` resolved at 50 steps but failed at 100, to check whether additional steps actively harm the agent. The dominant failure mode is exploration-path divergence (18 of 43): the two runs are independent executions that explore different trajectories, and the 50-step run happened to find better paths. Genuine self-sabotage (the agent producing a correct patch then degrading it) accounts for only 2 of 43 cases. The agent does not harm itself with more steps; it simply lacks a structured plan for converting extra budget into better patches.

7 Discussion

Reconciling prior findings. Lulla et al. (2026) evaluate focused pull requests where budgets are likely generous relative to task complexity; Gloaguen et al. (2026) evaluate complex benchmark tasks under default settings. Our budget experiment shows that the same guidance can appear beneficial, neutral, or harmful depending on available steps. Additionally, Gloaguen et al. (2026)'s context files are generated in a single LLM pass, while probe-and-refine guidance is iteratively refined through failure feedback, consistent with Zhu et al. (2026)'s finding that curated experience helps while unfiltered experience does not. To our knowledge, this is the first systematic variation of step budget across guidance conditions in the SWE-bench literature; existing studies report a single fixed budget without examining how it interacts with the guidance provided.

What the procedure learns. The qualitative evidence in Figure 3 is as important as the quantitative results. The procedure independently discovers test paths, inheritance chains, and middleware dependencies specific to each repository. These are operational details that a generic prompt could never specify. They are discovered through the feedback loop of failure and revision, which is why the procedure outperforms single-pass generation. The systematic content analysis (Table 2) reveals that the largest category of additions is procedural guardrails (47 %), such

Table 8: Unique solves at different budgets. Probe-and-refine dominates at 200 steps with 41 exclusive solves, more than the other two combined. The stable union (∼41 %) across budgets confirms the conditions explore genuinely different solution paths.

Condition	50 steps	100 steps	200 steps
`probe_refined`	20	29	**41**
`no_context`	25	21	18
`static_kb`	26	16	9
Union (all 3)	206 (41.2%)	207 (41.4%)	207 (41.4%)

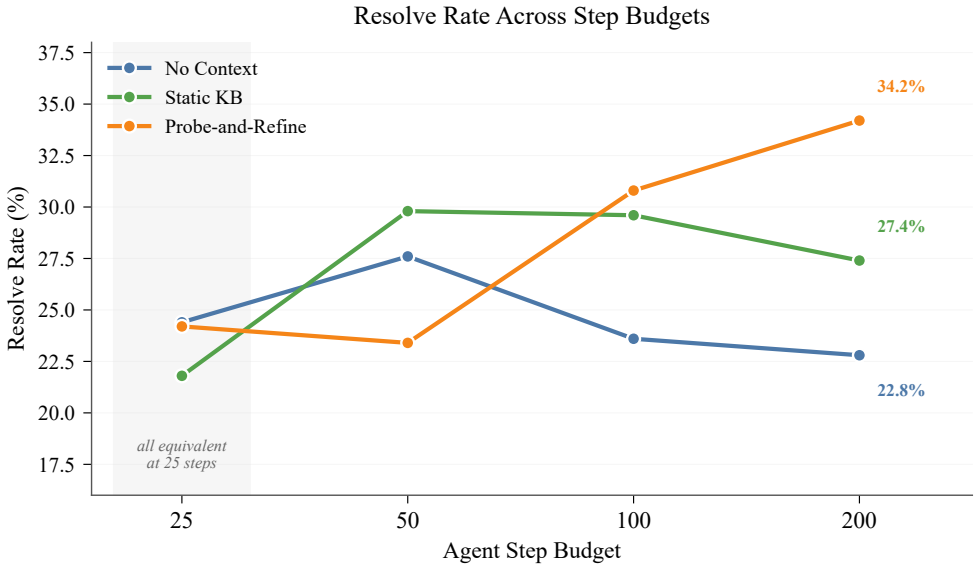

Figure 6: Resolve rate across step budgets. At 25 steps, all conditions are equivalent. As budget increases, the conditions progressively separate. The unguided baseline is flat at ∼23 % regardless of budget. Probe-and-refine is the only condition that continues improving beyond 100 steps. Error bars show 95 % Wilson CIs.

as "reproduce first, localize first, document before editing," suggesting the agent's main failure mode is jumping to fixes without sufficient diagnosis. The structural additions (30 %) encode repo-specific knowledge that makes these procedural rules actionable, and the quality gates (23 %) address output-quality failures like fabricated test results. This pattern holds across all 12 repositories, not just Django.

Simplicity as a feature. The procedure is composed entirely of single-shot LLM calls. There is no multi-step reasoning during tuning, no tool use, no gradient updates. The only ingredients are a repository's codebase, a model that can generate synthetic tasks and diagnose failures in its own attempted solutions, and a few iterations of editing a text file. That this is sufficient to significantly outperform both an unguided agent and a single-pass knowledge base suggests that the bottleneck in current coding agents may not be reasoning ability or context length but the quality of the instructions they are given.

Guidance makes compute productive. The coverage and budget findings together reveal a pattern worth stating explicitly: guidance does not make the agent a better programmer (precision is constant at ∼59 %), but it is what makes additional compute productive. The unguided baseline spends 2.2× more tokens at 200 steps than at 25 and gains nothing; probe-and-refine spends 2.6× more and converts the additional compute into a 10 pp improvement. Without a prescribed workflow, more steps produce more exploration but not better patches; guidance is the mechanism that turns compute into coverage.

Complementarity and ensembling. The union of all three conditions resolves 41.4 % of instances, 34 % more than the best single condition (Table 8). This gap suggests that an adaptive strategy selecting guidance depth based on repository or task characteristics could capture substantial further gains. Conversely, the remaining ∼59 % of instances

resist all three conditions, suggesting that breaking past this ceiling requires interventions beyond guidance, such as stronger models, longer context, or tree search.

Contamination robustness. Memorized patches would be solved regardless of guidance condition, appearing in the intersection of all three solve sets rather than as exclusive solves. The 41 unique solves from probe-and-refine are distributed across 9 of 12 repositories and mirror the overall difficulty distribution, suggesting the improvement comes from guidance content rather than memorization. We cannot rule out that guidance triggers recall of memorized solutions the unguided prompt fails to elicit, but the unique solves show no skew toward older or more-starred issues.

Absolute performance. Resolve rates (22–34 %) reflect a constrained pipeline: a 35B MoE model with 16k effective context and aggressive output truncation. Frontier systems achieve >55 % on SWE-bench Verified. The contribution is a controlled experiment showing that iteratively refined guidance significantly outperforms alternatives within the same pipeline.

8 Limitations

No repeated runs. Each condition is a single run. We estimate ∼4 pp of run-to-run variance based on the exploration-path divergence analysis in Section 6: of 43 instances where `no_context` resolved at 50 steps but not at 100, 18 were attributable to different exploration trajectories rather than systematic factors, implying that stochastic path selection accounts for roughly 4 % of outcomes. The probe-and-refine vs. no-context gap (11.4 pp, McNemar $p < 0.001$) is robust to this variance. The probe-and-refine vs. static-KB gap (6.8 pp, McNemar $p < 0.001$) is significant but closer to the noise floor. McNemar's test controls for instance-level pairing but not for run-level stochasticity; 3–5 repeated runs per condition would separate the systematic effect from exploration-path variance and are the highest-priority future work.

Guidance length confound. The refined guidance is 63 % longer on average than the static KB (Table 1), so we cannot disentangle content quality from prompt length. The content analysis (Table 2) suggests the added material is predominantly repo-specific knowledge rather than generic padding, and Gloaguen et al. (2026) find that longer LLM-generated context can *reduce* performance, but neither observation rules out a length effect. Two ablations would isolate this confound: padding `static_kb` with generic text to match the refined guidance's length, and truncating refined guidance to `static_kb`'s length. We did not run these ablations; until they are performed, the possibility that the model simply benefits from more prompt tokens remains open.

Single model and scaffold. Results are from one model with one custom pipeline. Frontier models may activate guidance benefits at lower step counts. Scaffolds with context summarization or tree-search may interact with guidance differently.

Probe sensitivity. We generate 10 probes per iteration at temperature 0.9 with deduplication, but do not analyze probe quality, diversity, or the sensitivity of the final guidance to different probe sets. Whether the procedure converges to similar guidance from different random seeds is an open question; the early-stop behavior (3–5 iterations) suggests some runs may be more productive than others.

Single benchmark with Django concentration. Findings are specific to SWE-bench Verified (500 instances), with Django accounting for 46 %. Table 4 shows the effect holds in both subsets, but the non-Django subset contains 11 repositories with as few as 1–2 instances each, making per-repo significance testing infeasible and limiting claims about cross-repository generality. The effect could be primarily a Django effect with modest support from other repositories; evaluation on benchmarks with more balanced repository distributions is needed.

9 Conclusion

We introduced probe-and-refine tuning, a lightweight procedure that iteratively transforms static repository knowledge into specialized operational guidance through ∼22 single-shot LLM calls per iteration: generate a batch of probes, attempt and judge each probe against expected behaviors, and aggregate diagnostics into mechanically applied guidance edits. On SWE-bench Verified, the resulting guidance achieves 34.2 % resolve rate (McNemar $p < 0.001$ vs. both baseline and static KB) and produces 41 unique solves distributed across 9 of 12 repositories. The mechanism is evaluation coverage: the guidance gives the agent a structured workflow that converts exploration into well-formed, evaluable patches more reliably, without improving the correctness of any individual evaluated patch.

The procedure is simple enough to be surprising. That a procedure composed entirely of single-shot LLM calls ($\sim$22 per iteration), with no tool use and no multi-step reasoning, can encode repo-specific operational knowledge sufficient to significantly outperform both an unguided agent and a static knowledge base suggests that the instructions given to coding agents matter at least as much as the agents' own reasoning capabilities.

For practitioners: invest in iteratively refined guidance for repositories where your agent will handle many issues, and ensure the step budget accommodates the prescribed workflow.

A McNemar's Test: Exact Derivation

Because each SWE-bench instance is evaluated under all three conditions, the outcomes are paired. McNemar's test (McNemar, 1947) compares the discordant pairs: for conditions A and B, let b be the number of instances A solves but B does not, and c the reverse. The test statistic is $\chi^2 = (b-c)^2/(b+c)$, distributed as χ^2_1 under the null hypothesis that A and B have equal solve rates. Intuitively, if A and B perform equally, instances where they disagree should split evenly (roughly $b \approx c$); a large imbalance indicates a real difference.

We can reconstruct the full three-way overlap from the reported data. Let P, N, S denote the solve sets for probe-refined, no-context, and static-KB respectively, with $|P| = 171$, $|N| = 114$, $|S| = 137$, $|P \cup N \cup S| = 207$, and exclusive counts $|P \text{ only}| = 41$, $|N \text{ only}| = 18$, $|S \text{ only}| = 9$.

Defining the pairwise-only overlaps $x = |P \cap N \setminus S|$, $y = |P \cap S \setminus N|$, $z = |N \cap S \setminus P|$, and $w = |P \cap N \cap S|$, the set-size constraints yield a fully determined system:

$$\begin{aligned} x + y + w &= 130 \quad (\text{from } |P| = 41 + x + y + w = 171) \\ x + z + w &= 96 \quad (\text{from } |N| = 18 + x + z + w = 114) \\ y + z + w &= 128 \quad (\text{from } |S| = 9 + y + z + w = 137) \\ x + y + z + w &= 139 \quad (\text{from } |P \cup N \cup S| = 68 + x + y + z + w = 207) \end{aligned}$$

Solving: $z = 139 - 130 = 9$, $y = 139 - 96 = 43$, $x = 139 - 128 = 11$, $w = 139 - 11 - 43 - 9 = 76$.

Probe-refined vs. no-context. $|P \cap N| = x + w = 87$. So $b = 171 - 87 = 84$ (probe solves, no-context does not) and $c = 114 - 87 = 27$ (no-context solves, probe does not). McNemar $\chi^2 = (84-27)^2/(84+27) = 3249/111 = 29.3$, $p < 0.001$.

Probe-refined vs. static-KB. $|P \cap S| = y + w = 119$. So $b = 171 - 119 = 52$ and $c = 137 - 119 = 18$. McNemar $\chi^2 = (52-18)^2/(52+18) = 1156/70 = 16.5$, $p < 0.001$.

Static-KB vs. no-context. $|N \cap S| = z + w = 85$. So $b = 114 - 85 = 29$ (no-context solves, static does not) and $c = 137 - 85 = 52$ (static solves, no-context does not). McNemar $\chi^2 = (29-52)^2/(29+52) = 529/81 = 6.5$, $p = 0.011$.

References

SWE-bench. SWE-bench Verified. 2024. `https://www.swebench.com/verified.html`.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? In *ICLR*, 2024.

Qwen Team. Qwen3.5-35B-A3B. 2026. `https://huggingface.co/Qwen/Qwen3.5-35B-A3B`.

Jai Lal Lulla, Seyedmoein Mohsenimofidi, Matthias Galster, Jie M. Zhang, Sebastian Baltes, and Christoph Treude. On the Impact of AGENTS.md Files on the Efficiency of AI Coding Agents. In *ICSE JAWs*, 2026. arXiv:2601.20404.

Thibaud Gloaguen, Niels Mündler, Mark Müller, Veselin Raychev, and Martin Vechev. Evaluating AGENTS.md: Are Repository-Level Context Files Helpful for Coding Agents? *arXiv:2602.11988*, 2026.

Aman Madaan et al. Self-Refine: Iterative Refinement with Self-Feedback. *NeurIPS*, 2023.

Noah Shinn, Federico Cassano, Emanuel Berman, Ashwin Gopinath, and Karthik Narasimhan. Reflexion: Language Agents with Verbal Reinforcement Learning. *NeurIPS*, 2023.

Shunyu Yao et al. ReAct: Synergizing Reasoning and Acting in Language Models. *ICLR*, 2023.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R. Narasimhan, and Ofir Press. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. *NeurIPS*, 2024.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying LLM-based Software Engineering Agents. *arXiv:2407.01489*, 2024.

Antonis Antoniades, Albert Örwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Wang. SWE-Search: Enhancing Software Agents with Monte Carlo Tree Search and Iterative Refinement. *ICLR*, 2025.

Siru Ouyang et al. RepoGraph: Enhancing AI Software Engineering with Repository-level Code Graph. *ICLR*, 2025. arXiv:2410.14684.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. AutoCodeRover: Autonomous Program Improvement. *ISSTA*, 2024.

Worawalan Chatlatanagulchai, Hao Li, Yutaro Kashiwa, et al. Agent READMEs: An Empirical Study of Context Files for Agentic Coding. *arXiv:2511.12884*, 2025.

Aristidis Vasilopoulos. Codified Context: Infrastructure for AI Agents in a Complex Codebase. *arXiv:2602.20478*, 2026.

Jared Zhu et al. SWE Context Bench: A Benchmark for Context Learning in Coding. *arXiv:2602.08316*, 2026.

Haoran Ye, Xuning He, Vincent Arak, Haonan Dong, and Guojie Song. Meta Context Engineering via Agentic Skill Evolution. *arXiv:2601.21557*, 2026.

Quinn McNemar. Note on the Sampling Error of the Difference Between Correlated Proportions or Percentages. *Psychometrika*, 12(2):153–157, 1947.

Alan Agresti. *Categorical Data Analysis*. Wiley, 3rd edition, 2013.

Thomas G. Dietterich. Approximate Statistical Tests for Comparing Supervised Classification Learning Algorithms. *Neural Computation*, 10(7):1895–1923, 1998.