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

A PREPRINT

Asa Shepard
Williams College
as66@williams.edu

April 19, 2026

ABSTRACT

AI coding agents need operational knowledge about a repository—which files house which subsystems, how to run the test suite, which workflows have historically led to wrong fixes—that is nowhere in the source code. Engineers maintain `AGENTS.md` files to supply this context as instructions for AI agents, but whether they help is contested: recent studies disagree on whether LLM-generated guidance improves or harms agent performance. We show that how guidance is produced is the decisive variable, and introduce *probe-and-refine tuning*: a procedure that uses synthetic bug-fix probes to iteratively diagnose and patch a repository’s guidance file through single-shot LLM calls, with no agent loop or tool use during tuning. On SWE-bench Verified across four independent trials with Qwen3.5-35B-A3B at 200 steps, probe-and-refine achieves 33.0% mean resolve rate vs. 28.3% for the static knowledge base used to initialize it and 25.5% for an unguided baseline ($p < 0.001$ for both contrasts). The improvement comes from coverage rather than precision: refined guidance produces evaluable patches for 14.5 pp more instances while per-patch precision remains statistically constant ($\sim 59\%$, $p = 0.119$), consistent with guidance helping the agent reach the correct file rather than making it a smarter programmer. A step-budget experiment shows guidance is what makes additional compute productive; a cross-model experiment with NVIDIA-Nemotron-3-Nano-30B-A3B finds that the tuning loop degrades when the model cannot generate sufficiently diagnostic output, though per-patch precision remains constant even then.

1 Introduction

AI coding agents—LLMs operating in a loop of reading code, running commands, and proposing edits—are increasingly being deployed against real software engineering tasks. Their effectiveness depends on more than just the model’s underlying capability. A repository is a body of accumulated decisions: subsystem boundaries, naming conventions, debugging workflows that experienced contributors have internalized but that are nowhere documented in the code itself. A model that has never worked in a particular codebase has to reconstruct this implicit knowledge on the fly, often by trial and error, and frequently lands in the wrong file or proposes a fix that breaks orthogonal subsystems.

Engineers have responded by maintaining repository-level context files—variously named `AGENTS.md`, `CLAUDE.md`, or similar—that document conventions, entry points, test-running instructions, and debugging strategies for use by AI agents. These files have proliferated rapidly: Chatlatanagulchai et al. (2025) survey 2,303 such files in active use across public repositories. Yet whether they 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, measuring efficiency rather than correctness. Gloaguen et al. (2026) find the opposite: LLM-generated context files reduce resolve rates by $\sim 3\%$ on SWE-bench Lite and AGENTbench, with agents following instructions literally even when those instructions are counterproductive. The disagreement leaves practitioners with no principled basis for deciding whether to invest in such guidance, what should go in it, or how to produce it.

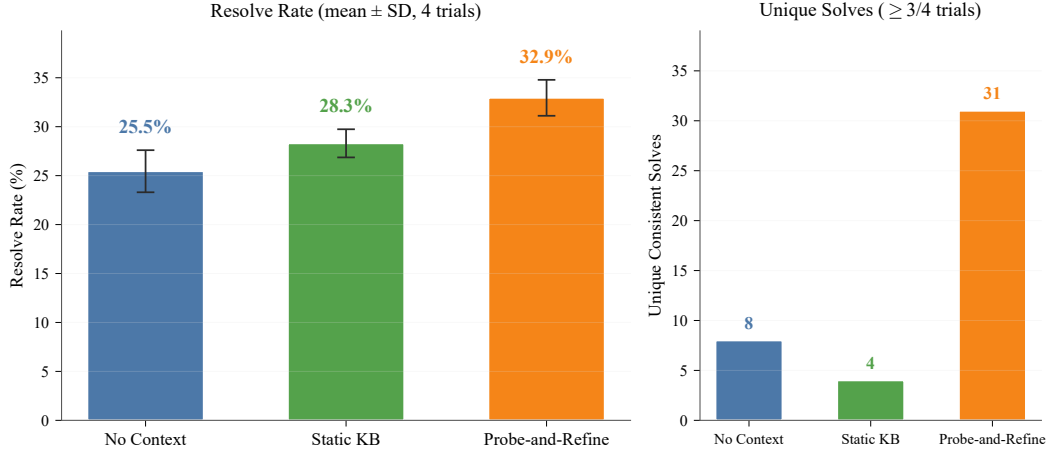


Figure 1: Mean resolve rate across four independent trials on SWE-bench Verified (500 instances per trial). Bars show four-trial means; error bars show ± 1 SD across trials. Probe-and-refine guidance significantly outperforms both the unguided baseline and the static knowledge base under a mixed-effects logistic regression on the pooled 6000 observations ($p < 0.001$ for both contrasts).

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. What an experienced contributor knows about a repository is operational and specific: *which* test file to run for a particular subsystem, *which* module to trace through when debugging a particular class of bug, *which* kinds of fixes to avoid because they have historically broken downstream consumers. This kind of knowledge cannot be produced by introspection alone; it has to be learned from observed failures. Yet the agent’s own instructions—the text that tells it how to approach a repository—have received surprisingly little systematic attention, with most improvement efforts focused on models, scaffolds, or context windows.

We introduce *probe-and-refine tuning*, a lightweight procedure that produces such operational guidance by repeatedly probing a candidate guidance file with synthetic bug-fix tasks, observing where it fails, and patching it. The procedure is composed entirely of single-shot LLM calls—no multi-step agent loop, no tool use, no reinforcement learning—and produces a compact (≤ 3000 -character) text artifact for each repository in a few iterations. The simplicity is intentional: we wanted to know whether the bottleneck in current coding agents is reasoning capability or instruction quality, and a procedure that does no agentic reasoning during tuning makes the answer cleaner. Our central finding is that *instruction quality matters substantially*, and that a feedback loop driven by synthetic failures is sufficient to produce instructions of the right kind.

The paper makes four contributions:

1. **Probe-and-refine tuning** (Section 3): a lightweight procedure composed entirely of single-shot LLM calls that refines a statically generated repository knowledge base into specialized guidance, achieving a mean 33.0 % resolve rate across four independent trials and 31 consistently unique solves.
2. **Mechanistic analysis** (Section 5): the improvement comes from evaluation coverage rather than patch correctness: the refined-guidance agent produces well-formed, evaluable patches for 14.5 pp more instances than the baseline while per-patch precision is statistically constant ($\sim 59\%$, $p = 0.119$).
3. **Budget moderation** (Section 7): a sixteen-cell experiment showing that step budget moderates the effect of guidance, with different guidance types activating at different thresholds and unstructured exploration saturating quickly regardless of budget.
4. **Cross-model boundary conditions** (Section 8): a capacity-constrained model cannot sustain the tuning loop, and guidance calibrated for one model’s behavioral profile actively destabilizes a different model’s agent loop while per-patch precision remains constant.

The remainder of the paper presents the procedure, the four-trial replication that establishes these claims quantitatively, the qualitative localization analysis that explains them, the budget experiment that situates them, and a cross-model analysis that identifies the procedure’s model-fit requirements.

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).

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 ≤ 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

At a high level, the experiment compares how a coding agent performs on SWE-bench Verified under three different repository-context conditions: with no guidance, with a two-layer static knowledge base comprising tree-sitter-assisted parsing of the repository structure and one-shot LLM-generated generic guidance, and with guidance produced by our probe-and-refine procedure. The procedure itself is the only novel system component; everything else—the agent scaffold, the evaluation harness—is held fixed across conditions in the main experiment so that any performance difference is attributable to the guidance. The four pipeline stages we describe in this section are: building the repository context, running probe-and-refine tuning to produce the refined-guidance condition, generating patches with an interactive coding agent, and evaluating those patches with 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 generation¹. Command outputs are truncated to ~ 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 9.

3.2 Context Conditions

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

¹Code 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

`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.

`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 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, ~ 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.

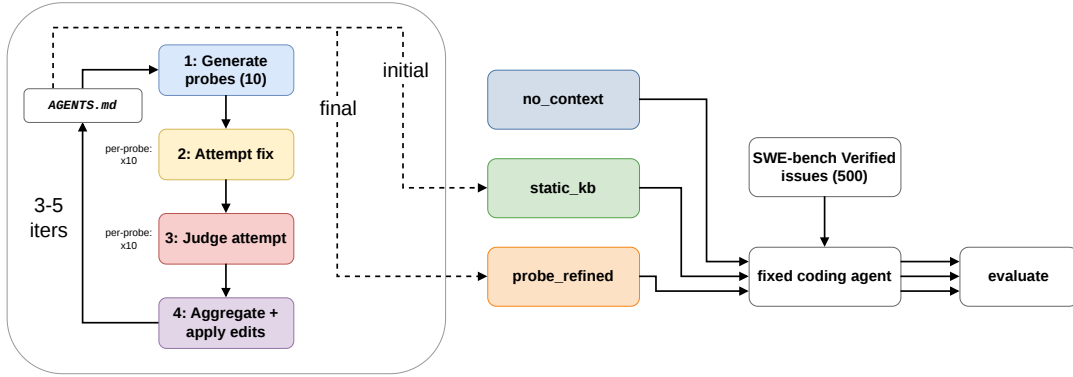


Figure 2: Probe-and-refine tuning pipeline. The `static_kb` artifact feeds both the `static_kb` condition directly and the refinement loop, which transforms it into the refined guidance using synthetic probes and single-shot diagnosis. No SWE-bench evaluation instances are used during refinement. All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

- 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.
- 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 tends to add a reproduce-first diagnostic workflow, with repo-specific navigation and quality-gate strategies filling in subsequent phases. 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 ~ 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.

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”

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. Mean fallback rates across the four trials differ substantially across conditions (14.8% for probe-and-refine vs. 30.8% for static-KB and 25.6% for no-context at 200 steps), so the fallback pathway is not a uniform noise source; the guided agent more often succeeds within its main loop.

Table 3: Pinned repository commits. Django accounts for 46 % of instances; we report results separately for both subsets in Table 7.

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	3acc3d3f	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

3.5 Experimental Design

We evaluate all three conditions at 200 agent steps on 500 SWE-bench Verified instances across four independent trials. A secondary experiment additionally varies step budgets (25, 50, 100) to study how guidance interacts with available compute (Section 7); the budget-variation runs were conducted as single trials at each non-200 budget, with only the 200-step condition replicated four times. 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). The four trials at 200 steps were conducted on the same hardware with identical hyperparameters, conducted independently, allowing us to estimate run-to-run variance and report effect sizes with appropriate uncertainty rather than relying on single-run point estimates.

Statistical tests. With four trials of 500 paired instances each at 200 steps, we have two natural sources of variance: instance-level (some instances are harder than others) and trial-level (run-to-run stochasticity in agent exploration). Our primary significance test is a mixed-effects logistic regression of the form $\text{resolved} \sim \text{condition} + (1 | \text{instance}) + (1 | \text{trial})$, fit via `lme4::glmer` on the pooled $4 \times 500 \times 3 = 6000$ observations. This model accounts for both the within-trial pairing structure and across-trial variance simultaneously, and is the appropriate framework for repeated-measures binary outcomes (Bates et al., 2015; Agresti, 2013). We supplement with within-trial McNemar’s tests (McNemar, 1947) for each pairwise comparison in each trial, providing a per-trial paired analysis (Dietterich, 1998). For descriptive comparisons we report mean \pm SD across the four trials and 95 % Wilson confidence intervals on per-trial proportions.

4 Results

Across four independent trials, probe-and-refine guidance achieves a mean resolve rate of 33.0 % (SD 1.8 pp), compared with 28.3 % (SD 1.4 pp) for static KB and 25.5 % (SD 2.2 pp) for no context (Table 4, Figure 1). The ordering probe-refined $>$ static $>$ no-context holds in 4/4 trials with no exceptions, for both resolve rate and coverage. Per-trial Wilson 95 % CIs on the resolve rates do not overlap between probe-refined and no-context in three of the four trials (T1, T3, T4); in T2 the CIs overlap by a small margin, though the within-trial McNemar test remains significant ($p < 0.001$) due to the paired structure.

Hierarchical model. A mixed-effects logistic regression on the pooled 6000 observations, with instance and trial as crossed random effects, confirms both key contrasts at $p < 0.001$ (Table 5). The trial random effect variance is estimated at approximately zero (singular fit), indicating that run-to-run drift at the trial level is negligible relative to instance-level variance ($\sigma_{\text{instance}}^2 = 9.12$); the variance we observe across trials is dominated by which specific instances each run happened to resolve, not by systematic differences between runs.

Per-trial McNemar tests. We supplement the hierarchical model with per-trial McNemar’s tests on the within-trial paired observations. Probe-refined vs. no-context is significant in 4/4 trials ($p < 0.001$ in every trial). Probe-refined vs. static-KB is significant in 4/4 trials ($p < 0.05$ in every trial; $p < 0.001$ in T1 and T4). Static-KB vs. no-context is significant in 3/4 trials, borderline in T3.

Table 4: Per-trial resolve rate, coverage, and precision by condition. The ordering probe-refined > static > no-context holds in 4/4 trials for both resolve rate and coverage. Precision is approximately constant across conditions in every trial.

Trial	Condition	Resolve	Coverage	Precision
T1	no_context	22.8 %	37.4 %	61.0 %
	static_kb	27.4 %	47.2 %	58.1 %
	probe_refined	34.2 %	57.4 %	59.6 %
T2	no_context	27.8 %	43.6 %	63.8 %
	static_kb	30.2 %	52.2 %	57.9 %
	probe_refined	30.6 %	54.4 %	56.2 %
T3	no_context	24.8 %	43.2 %	57.4 %
	static_kb	27.0 %	51.0 %	52.9 %
	probe_refined	32.4 %	55.0 %	58.9 %
T4	no_context	26.4 %	42.8 %	61.7 %
	static_kb	28.6 %	52.0 %	55.0 %
	probe_refined	34.6 %	58.0 %	59.7 %
Mean \pm SD	no_context	25.5 \pm 2.2 %	41.7 \pm 2.9 %	61.0 \pm 2.6 %
	static_kb	28.3 \pm 1.4 %	50.6 \pm 2.3 %	56.0 \pm 2.5 %
	probe_refined	33.0 \pm 1.8 %	56.2 \pm 1.8 %	58.6 \pm 1.6 %

Table 5: Mixed-effects logistic regression results: resolved \sim condition + (1 | instance) + (1 | trial), fit on pooled $n=6000$ observations.

Contrast	β	Odds Ratio	p
probe_refined vs. no_context	+0.748	2.11	< 0.001
static_kb vs. no_context	+0.293	1.34	0.004
probe_refined vs. static_kb	+0.456	1.58	< 0.001

Decomposing the improvement. The mean gain from no-context to probe-and-refine across the four trials is 7.5 pp. Of this, 2.8 pp comes from the static knowledge base (structural layer plus generic advice), and 4.7 pp from the iterative refinement process. The structural layer thus accounts for roughly 37 % 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 has the lowest fallback rate (mean 14.8 %, vs. 25.6 % for no-context and 30.8 % for static-KB across the four trials), meaning it most often produces patches through the coding agent’s own loop rather than the single-shot fallback. It also consumes roughly 56 % more prompt tokens than the unguided baseline, but resolves 29 % more instances—more compute per instance, but proportionally more correct patches. Per-repository variation is substantial; for example, on matplotlib, probe-and-refine achieves 76.9 % precision at 100 steps (single-trial measurement), the largest per-repo gap in the study.

Unique solves and consistency. Across the four trials, instances can be characterized by how consistently each condition resolves them. At a strict threshold of ≥ 3 of 4 trials, probe_refined resolves 31 instances that neither no_context nor static_kb resolves consistently, compared with 8 unique-consistent solves for no_context and 4 for static_kb (Table 6). 65 instances are resolved consistently by all three conditions (the “easy core”), and 342 are never resolved by any condition in ≥ 3 trials (the “hard core” where guidance does not help). The middle 93 instances are where guidance matters, and probe-refined dominates this contested middle.

Django vs. non-Django performance. Because Django accounts for 46 % of instances, we verify that results are not driven solely by this repository (Table 7). The probe-and-refine advantage is present in both subsets and roughly proportional, with no evidence that the effect is Django-specific.

Table 6: Unique solves by condition and step budget. The 50- and 100-step columns are single-trial counts; the 200-step column uses the four-trial consistent-solve threshold (≥ 3 of 4 trials). Probe-refined dominates the contested middle at the highest budget while all conditions explore different solution paths at lower budgets.

Condition	50 steps	100 steps	200 steps (4-trial)	Easy / Hard core
probe_refined	20	29	31	
no_context	25	21	8	65 / 342
static_kb	26	16	4	

Table 7: Mean resolve rates across four trials by repository subset at 200 steps. The probe-and-refine advantage is present in both subsets.

Subset	no_context	static_kb	probe_refined
Django ($n=231$)	27.7 %	31.2 %	35.3 %
Non-Django ($n=269$)	23.5 %	25.8 %	30.9 %
All ($n=500$)	25.5 %	28.3 %	33.0 %

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. A mixed-effects logistic regression on the evaluated subset only (i.e., resolved \sim condition + (1 | instance) + (1 | trial) restricted to evaluable patches, $n=2971$) finds no significant effect of condition on precision: likelihood-ratio test $\chi^2(2) = 4.26$, $p = 0.119$. Pairwise contrasts on the evaluated subset are also non-significant (probe-refined vs. no-context: $p = 0.47$; static vs. no-context: $p = 0.078$). Conditional on producing a patch the harness can evaluate, the three conditions resolve issues at approximately the same rate.

Precision also holds constant across step budgets in the single-trial runs: 55–60 % at 50 steps and 57–60 % at 100 steps across all three conditions, consistent with the four-trial finding at 200 steps.

Probe-and-refine produces evaluable patches for 56.2 % of instances on average vs. 41.7 % for no-context (Figure 4)—a 14.5 pp gap. At probe-and-refine’s observed precision (58.6 %), 72 additional evaluable patches per trial should account for ~ 42 additional resolves; the observed gap is 38. Coverage accounts for approximately 88 % of the resolve-rate gap.

The coverage ordering probe-refined $>$ static $>$ no-context replicates in 4/4 trials with low SDs (1.8 pp for probe-refined, 2.3 pp for static, 2.9 pp for no-context); notably, the guided conditions are more stable across trials than the unguided baseline.

What “evaluable” means. The SWE-bench harness rejects malformed patches, patches that do not apply cleanly, or patches that crash the test suite before assertions can be checked; reaching evaluation therefore requires syntactically valid, well-scoped, correctly targeted output. The coverage gap therefore 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 across the four trials. Agent-loop patches resolve at 60.1 % when they reach evaluation (1,729 of 2,875 pooled across conditions and trials); fallback patches resolve at 5.2 % (5 of 96). The fallback pathway is effectively a last-ditch probe rather than a recoverable path to a correct patch: across the four trials, only five fallback-produced patches resolved at all. This has two implications. First, the precision constancy across conditions (Table 8) is driven almost entirely by agent-loop patches. Second, conditions that trigger the fallback less often (probe-and-refine at 14.8 % vs. static-KB at 30.8 % vs. no-context at 25.6 %) are not merely avoiding a slightly worse pathway; they are avoiding a pathway that almost never produces correct patches. This reinforces the coverage interpretation: what matters is whether the agent can produce an evaluable patch within its main loop at all.

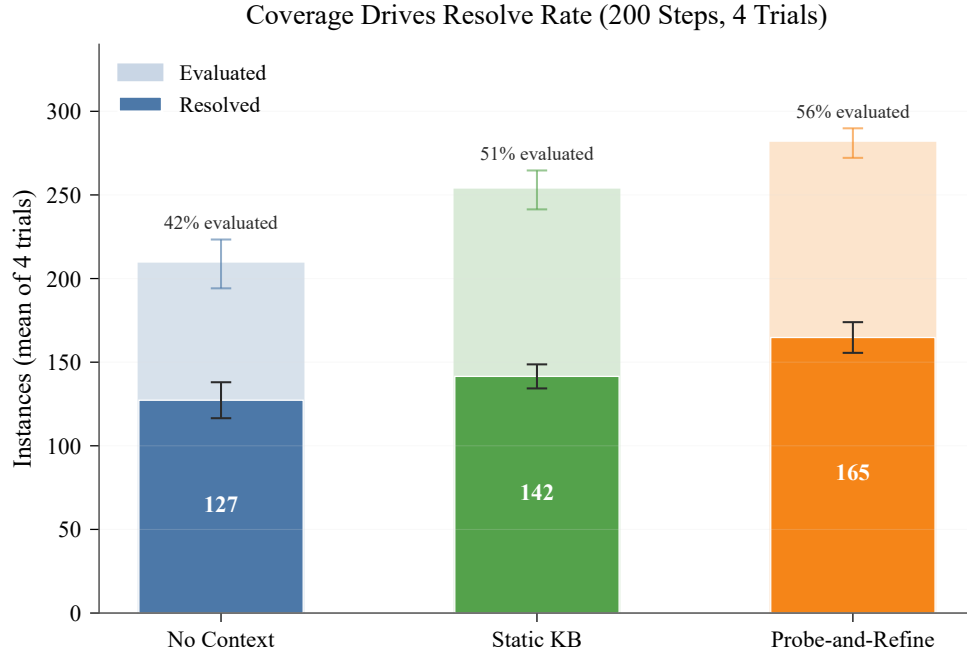


Figure 4: Mean evaluation coverage across four trials at 200 steps. Probe-and-refine produces evaluable patches for 56.2 % of instances on average (SD 1.8 pp) vs. 41.7 % (SD 2.9 pp) for no context. Precision ($\sim 59\%$) is similar across conditions and not statistically distinguishable in a hierarchical model on the evaluated subset.

Table 8: Mean precision (resolved / evaluated) across four trials at 200 steps. Differences across conditions are not statistically significant in a hierarchical model on the evaluated subset ($p = 0.119$).

Condition	Precision (mean \pm SD)
no_context	61.0 \pm 2.6 %
static_kb	56.0 \pm 2.5 %
probe_refined	58.6 \pm 1.6 %

5.2 The Agent Uses Late Steps Productively

The step distribution (Figure 5) shows where the coverage advantage originates. Across the four trials, the unguided agent produces only 4.0 % of its agent-loop patches after step 100, vs. 9.6 % for static KB and 12.6 % for probe-and-refine—a roughly $3\times$ gap in late-step productivity between probe-and-refine and no-context. 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 $\sim 59\%$ precision rate as earlier patches.

6 A Localization Interpretation

To understand what kind of instances refined guidance helps with, we examined the 31 instances that `probe_refined` resolves consistently (≥ 3 of 4 trials) and that neither `no_context` nor `static_kb` resolves consistently—the cleanest available signal of what probe-refined guidance uniquely contributes.

The unique solves are not the hard instances. Contrary to the natural prediction that additional guidance would help on harder bugs, the 31 probe-refined-only consistent solves are easier than the benchmark average. Median patch size is 5 added lines (vs. 4 across the full benchmark); 13 % are multi-file (vs. 14 %); 13 % include a traceback in the problem statement (vs. 14 %); 45 % are SWE-bench-rated as < 15 minute fixes (vs. 39 %) and only one exceeds one hour. Roughly a third are small additive feature requests (e.g., “expose `warm_start`”, “add `fill_value`”, “make

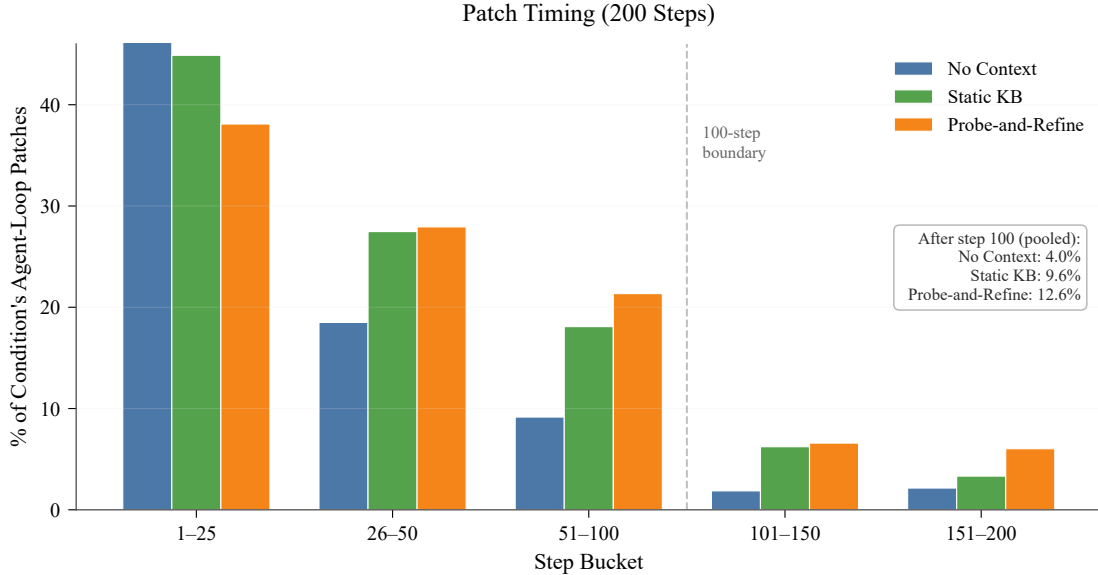


Figure 5: Patch timing at 200 steps, pooled across four trials. The unguided agent produces the bulk of its patches in the first 25 steps and very few thereafter; probe-and-refine produces a substantially larger fraction of its patches after step 100.

element_id optional in json_script”) where the task is not writing a complex fix but locating the correct plumbing point for a small change.

The unique solves share a localization mismatch. Reading the 31 problem statements qualitatively, a single pattern dominates: the problem statement names a user-facing API surface (e.g., `IsolationForest`, `FITSDiff`, `TruncDate`, `DataArray.quantile`, `strip_accents_unicode`, `SubFigure.legend`) while the actual fix lives in a non-obvious internal module (BaseBagging plumbing inside `sklearn.ensemble._iforest`, VLA column comparison inside `astropy.io.fits.diff`, `TruncBase.as_sql`, `xarray.core.variable`, NFKD handling inside `sklearn.feature_extraction.text`, the `Legend.__init__` parent-type check, etc.). A model searching by named symbol is likely to land in the wrong file. Refined guidance, which encodes repo-specific subsystem structure and tracing instructions, may point the model to the correct module. More broadly, even when the guidance does not name the specific file implicated in an issue, exposure to concrete paths and module references may shift the agent toward path-oriented search strategies across the repository as a whole—a hypothesis the current data cannot directly test but that would explain partial localization benefits outside the cleanly matched cases.

The repository distribution supports the localization hypothesis. If probe-refined guidance functions primarily as a localization aid, repositories with more predictable layouts—where symbol names map cleanly to file locations from pretraining—should benefit less than repositories with idiosyncratic internal organization. This is what we observe (Figure 6). Django, with its standardized MVC layout (`django/db/models/`, `django/views/`, `django/template/`), is underrepresented among probe-refined-only consistent solves (9 observed vs. 14.3 expected from base rate, ratio 0.63). Scientific and numerical libraries with flatter or more idiosyncratic layouts—`scikit-learn` (2.02 \times), `xarray` (2.20 \times), `matplotlib` (1.42 \times), `sympy` (1.29 \times)—are over-represented. Pytest, which has a flat and consistent module structure, contributes zero of 1.2 expected probe-refined-only consistent solves.

Synthesis with the coverage finding. The localization hypothesis is consistent with both the coverage and precision results. Refined guidance appears to help the agent reach the correct file for small mechanical fixes; this would produce the coverage gap we observe, because patches that target the wrong file are rejected by the harness or fail to apply cleanly. Conditional on reaching the right file, the fix is small and either works or does not, which is why per-patch precision is constant across conditions. The pattern is consistent with refined guidance helping the agent find the right file rather than making it a smarter programmer—helping it find the haystack so that the needle, when small, can be threaded. This interpretation also explains why the procedure does not break past the $\sim 58\%$ any-trial union ceiling (and a tighter 32% consistent-solve ceiling): instances requiring genuinely complex multi-file refactors or hard algorithmic insight are not in the population that localization helps.

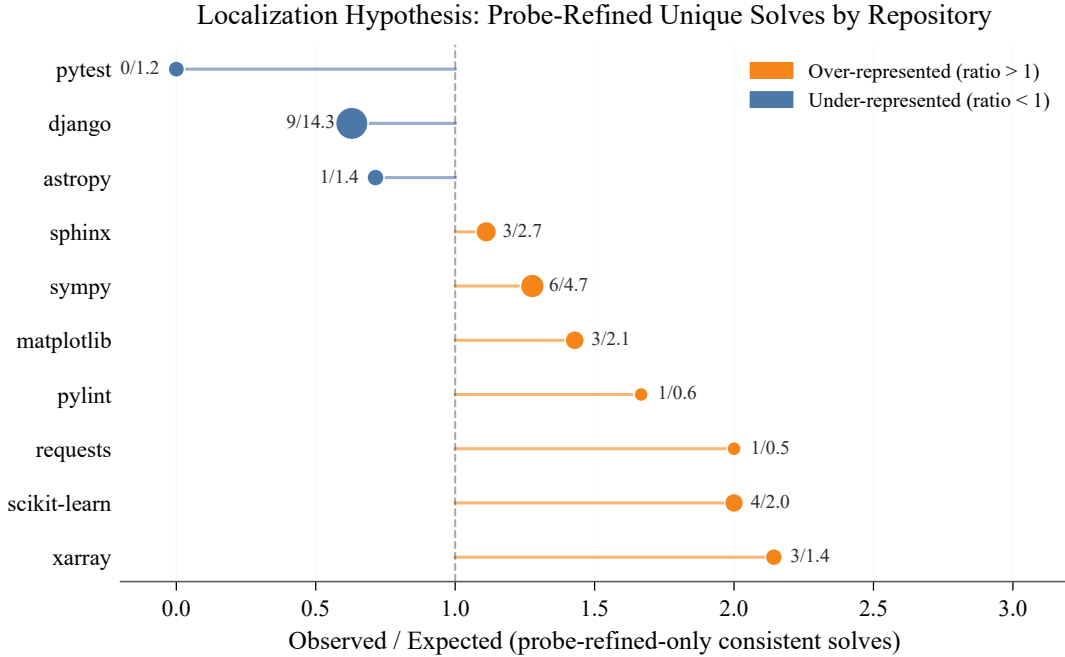


Figure 6: Probe-refined-only consistent solves per repository, expressed as a ratio of observed to expected (base-rate). Dot size is proportional to repository size; larger dots represent more reliable estimates. Repositories with predictable layouts (django, pytest) fall below the reference line; those with idiosyncratic internal organization (xarray, scikit-learn) exceed it.

Table 9: Resolve rates across step budgets. The 200-step results are the four-trial means reported in Section 4; the 25, 50, and 100-step results are single-trial measurements.

Condition	25	50	100	200 (mean)
no_context	24.4 %	27.6 %	23.6 %	25.5 %
static_kb	21.8 %	29.8 %	29.6 %	28.3 %
probe_refined	24.2 %	23.4 %	30.8 %	33.0 %

7 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 9). The 200-step condition was replicated four times (Section 4); the 25, 50, and 100-step conditions are single-trial measurements, so the patterns we report across budgets should be read as descriptive rather than as replicated effects.

Three patterns are visible in Figure 7. First, all conditions are equivalent at 25 steps (21.8–24.4 %, all pairwise $p > 0.3$ via single-trial McNemar), 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 ($p = 0.004$, single-trial McNemar) then plateaus, while probe-and-refine is flat from 25 to 50 then rises from 23.4 % at 50 to 30.8 % at 100 and continues to a four-trial mean of 33.0 % at 200. Third, the unguided baseline is flat at 22.8–27.6 % across all four budgets ($p = 0.551$ for 25 vs. 200, single-trial McNemar), consuming roughly 2.3× more tokens at 200 as at 25 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, while static KB jumps from 21.8 % to 29.8 %. At 50 steps, probe-and-refine actually underperforms static KB ($p = 0.022$ in the single-trial McNemar test). We attribute this to a workflow-budget mismatch: the reproduce-trace-patch workflow prescribed 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

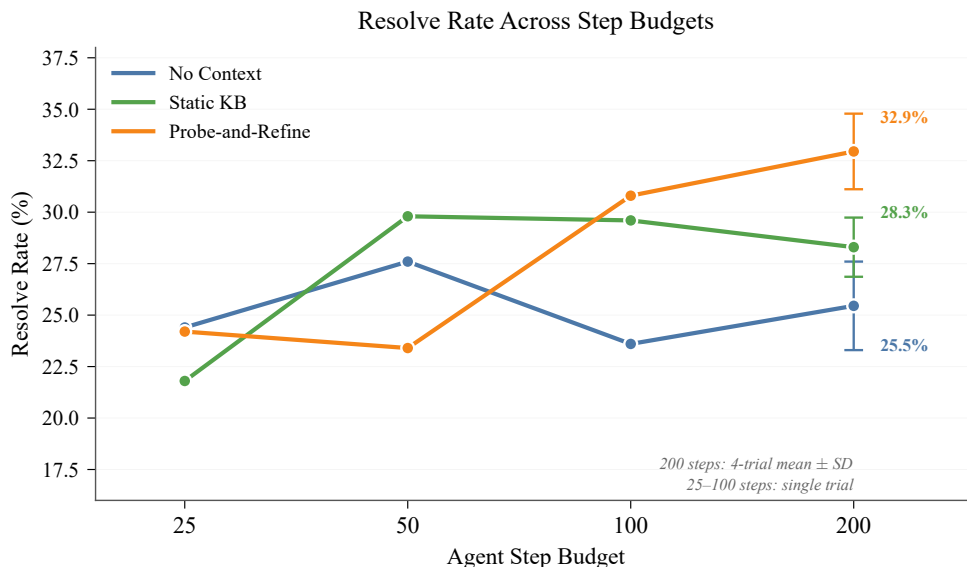


Figure 7: Resolve rate across step budgets. At 25 steps, all conditions are equivalent. As budget increases, the conditions progressively separate. The unguided baseline is flat at $\sim 25\%$ regardless of budget. Probe-and-refine is the only condition that continues improving beyond 100 steps. The 200-step point is the four-trial mean with ± 1 SD; lower-budget points are single-trial.

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 $2.3\times$ more compute at 200 steps than at 25 and gets nothing for it, while probe-and-refine spends $2.8\times$ more and converts the additional compute into a 7–10 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.

Toward a causal test of localization. A direct test of the localization hypothesis (Section 6) would compare `probe_refined` guidance against a variant with the file-structure portions stripped out, leaving only the procedural and quality-gate advice; if the localization hypothesis is correct, this stripped variant should regress toward `static_kb` performance while remaining above `no_context`. We did not run this ablation in the present work but identify it as the highest-priority follow-up experiment.

8 Model Capacity as a Boundary Condition

To test whether the procedure generalizes beyond Qwen3.5-35B-A3B, we ran the full experiment with NVIDIA-Nemotron-3-Nano-30B-A3B (Nemotron; NVIDIA 2026), a hybrid Mamba-Transformer mixture-of-experts model

Table 10: Tuning loop output by model. Nemotron’s loop degrades after the first iteration; five of twelve repositories produce zero new probes and zero edits in later iterations (average version reached: 3.8 of 5). Qwen generates 341 total probes and 175 edits versus Nemotron’s 165 and 78, reaching an average of 4.5 of 5 iterations.

Metric	Nemotron	Qwen
Avg. version reached	3.8 / 5	4.5 / 5
Total probes generated	165	341
Total edits applied	78	175
Avg. chars per repo	2,295	2,753
Growth over <code>static_kb</code>	+608 chars	+1,065 chars

Table 11: Resolve rates on SWE-bench Verified (500 instances, 200 steps). Qwen results are four-trial means; Nemotron results are single-trial. The two models respond oppositely to the same guidance conditions.

Condition	Qwen (mean, $n=4$)	Nemotron (single trial)
<code>no_context</code>	25.5 %	28.4 %
<code>static_kb</code>	28.3 %	24.6 %
<code>probe_refined</code>	33.0 %	27.0 %

with $\sim 3.5B$ active parameters—similar to Qwen3.5-35B-A3B’s 3B active parameters, but a different architecture and training profile. We used the same scaffold, the same 500 SWE-bench Verified instances, and the same evaluation harness. All Nemotron experiments were conducted as single trials; the four Qwen trials established that trial-level variance is negligible at $n=500$, so a single trial is sufficient to answer the qualitative question of whether the procedure generalizes.

8.1 The Tuning Loop Cannot Sustain on a Capacity-Constrained Model

The probe-and-refine loop requires three capabilities: generating diverse synthetic bugs, producing attempted patches rich enough to diagnose, and judging those attempts with enough specificity to propose actionable edits. Nemotron satisfies the first but struggles with the second and third. Its diagnostic output is too thin for the judge step to extract signal, so the loop degrades: five of twelve repositories produce zero new probes and zero edits after the first iteration, and Nemotron reaches an average of 3.8 of 5 iterations versus Qwen’s 4.5 (Table 10). Matplotlib, flask, pytest, xarray, and sympy all plateau at iteration 2 or earlier, with sympy’s guidance text essentially unchanged from its initial version.

Qwen’s guidance contains concrete, repo-specific strategies—“trace `_draw_3d_patch`, `_plot_surface`, and zorder calculation paths” for matplotlib—while Nemotron’s degraded loop produces generic elaborations of its initial boilerplate: “list candidate files and functions.” The gap suggests a model-fit dependency: probe-and-refine appears to require a model that can generate diagnostic output rich enough for the judge step to act on. The loop prompts were developed around Qwen’s output format, so Nemotron’s weaker loop participation may reflect prompt mismatch as much as raw capacity—a Nemotron-adapted prompt design remains untested.

8.2 Static Guidance Hurts a Capacity-Constrained Model

Nemotron self-tuned results show a pattern opposite to Qwen’s (Table 11): all guidance conditions underperform the unguided baseline, and the ordering is no-context > probe-refined > static-KB. Compare to Qwen, where the same guidance text monotonically improves performance and static-KB alone adds 2.8 pp over baseline.

The 3.8 pp gap between no-context and static-KB on Nemotron has two roughly equal contributing factors (~ 9 and ~ 10 instances each). On coverage: with static-KB guidance, Nemotron produces 328 agent-loop patches vs. 345 for no-context (17 fewer) and generates 131 non-actionable instances vs. 107 (24 more); guidance adds ~ 500 tokens per turn, increasing the probability of mode-switching from tool calls to prose. On patch quality: among the 261 instances where both conditions produce a patch, no-context resolves 44.1 % vs. static-KB’s 39.8 %; the guidance instructs the agent to “keep the first patch minimal,” and Nemotron interprets this literally—for `django-11276` it adds the import without connecting it, for `django-14238` it produces a shallow one-liner, for `sympy-11618` it imports `zip_longest` without using it. The root cause is instruction-following interference: Nemotron’s different behavioral profile leaves it more susceptible to behavioral steering by prescriptive language, shifting it from executing to deliberating. Qwen absorbs the same guidance as context while maintaining action-oriented behavior; Nemotron cannot. The probe-refined condition partially recovers over static-KB (27.0 % vs. 24.6 %), likely because the degraded loop removes some of the most harmful prescriptive language rather than adding useful repo-specific content.

Table 12: Agent behavior under Qwen-to-Nemotron cross-model transfer vs. Nemotron self-tuned baseline. Qwen’s guidance triggers compliance by analysis rather than action in Nemotron, collapsing agent-loop patch production and generating a large fallback cascade. Per-patch precision when the loop completes is unchanged (bottom row).

Metric	Qwen guidance	Nemotron baseline
Resolved	66 (13.2 %)	135 (27.0 %)
Agent-loop patches produced	174	351
Fallback patches triggered	324	3
Avg. steps / instance	81.0	39.2
Instances hitting max steps	83	11
Eval errors (patch-apply failures)	246	14
No-command prose events	2,141	1,252
Repeated-command stall instances	172	41
Agent-loop precision (resolved/patch)	37.9 %	38.5 %

8.3 Cross-Model Guidance Transfer Is Catastrophic

To test whether high-quality guidance produced by a capable model can transfer to a less capable one, we applied Qwen’s probe-refined guidance to Nemotron without modification (Table 12). The result is a collapse: 66 resolved instances (13.2 %), compared with 135 (27.0 %) for Nemotron’s own probe-refined baseline and 142 (28.4 %) for no-context.

The mechanism is compliance by analysis rather than action: Nemotron reads Qwen’s detailed prescriptive guidance and writes analysis of what it would do rather than running commands—71 % more no-command prose events and 4× the stall rate. This triggers a fallback cascade: when the agent loop produces no patch, the single-shot fallback fires carrying 81 steps of accumulated but unfocused context; the fallback then generates patches that are long but malformed (mean 901 characters versus 280 for the baseline), 243 of 246 evaluation errors are patch-apply failures originating from these fallback patches, and none of the 324 fallback patches resolve the target issue.² Crucially, per-patch precision when the agent loop completes is unchanged: 37.9 % under Qwen’s guidance vs. 38.5 % for Nemotron baseline ($p > 0.5$, Fisher’s exact test). The guidance breaks the agent’s control loop, producing far fewer evaluable patches, without touching the quality of the patches it does produce. This is the precision-constancy finding from Section 5.1 replicating from a completely unexpected direction: across models, across guidance conditions, and under adversarial transfer, the fraction of patches that resolve once they reach evaluation is stable. The catastrophe is entirely in coverage.

These results establish that guidance encodes model-specific behavioral calibration, not transferable repository knowledge. The gains probe-and-refine produces on Qwen reflect guidance tailored to Qwen’s specific failure modes; the same text is actively harmful to a model with a different behavioral profile. Whether guidance explicitly designed for cross-model use—or tuned to avoid model-specific behavioral cues—could support positive transfer is an open question; this experiment tests only incidental transfer, and a targeted multi-model tuning regime might not trigger the same failure mode.

9 Discussion

Instruction quality as a first-order variable. The central claim of this paper is that the quality of the instructions given to a coding agent is a first-order determinant of its reliability—comparable in magnitude to choices about model capability or step budget within a fixed scaffold. This reframes a question that has received surprisingly little systematic attention: while models, context windows, and scaffolds have all been subject to careful ablation, the content of the agent’s own instructions has rarely been varied or optimized. 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 2.3× more tokens at 200 steps than at 25 and gains nothing; probe-and-refine converts the additional compute into a 7–10 pp improvement. The 42 % of instances resistant to all three conditions at the any-trial threshold are unlikely to be reached by guidance improvements alone; these appear to require stronger reasoning or longer context rather than better localization.

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

²Eval health is intact: 198 of 441 submitted patches passed clean harness evaluation; the 246 errors are real malformed patches, not harness failures.

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. Critically, the Nemotron self-tuned experiment provides a within-study replication of Gloaguen et al. (2026)’s finding: the same `static_kb` guidance that adds 2.8 pp to Qwen’s resolve rate reduces Nemotron’s by 3.8 pp. The disagreement between prior studies may reflect differences in the models evaluated as much as differences in how guidance was produced or what tasks were attempted. 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.

Model capacity as a moderating variable. Practitioners should treat guidance complexity as a three-way match: workflow depth, step budget, and model capacity must all be commensurate for probe-and-refine to help rather than harm. A capacity-constrained model cannot sustain the feedback loop (Section 8), and prescriptive guidance calibrated for a more capable model triggers mode-switching rather than execution.

Guidance is model-specific. The cross-model transfer experiment (Section 8) demonstrates that guidance encodes model-specific behavioral calibration. Practitioners should tune guidance with the model that will consume it.

Contamination robustness. The 31 unique consistent solves from probe-and-refine are distributed across 9 of 12 repositories (Figure 6) 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 (22–35 %) 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.

10 Limitations

Variance characterization. Each condition was run in four independent trials at 200 steps, allowing us to characterize but not eliminate run-to-run variance. Additional trials would tighten confidence intervals further but are unlikely to change conclusions: the hierarchical model already gives $p < 0.001$ on the key contrasts at $n=4$ trials. The lower-budget runs (25, 50, 100 steps) in Section 7 are single-trial measurements and lack the same uncertainty quantification.

Guidance length confound. The refined guidance is 63 % longer on average than the static KB (Table 1), so we cannot fully disentangle content quality from prompt length. The content analysis (Table 2) shows that the added material is predominantly repo-specific knowledge—procedural guardrails, structural references, and quality gates—rather than generic padding. Furthermore, Gloaguen et al. (2026) demonstrate that longer LLM-generated context files can actively *reduce* performance, which is the opposite of what a pure length effect would predict. The localization analysis (Section 6) provides additional support: the instances probe-refined guidance uniquely resolves are concentrated in repositories where naive symbol-based localization fails, a pattern that generic-text padding would not produce. Two ablations would isolate this confound completely (padding `static_kb` to match refined-guidance length, and truncating refined guidance to `static_kb` length); we did not run these due to compute constraints, and the possibility that the model simply benefits from additional prompt tokens remains open.

Single model with demonstrated capability. The probe-and-refine procedure has been demonstrated on one model, Qwen3.5-35B-A3B. A cross-model experiment with Nemotron suggests the procedure has model-fit requirements: the model must generate diagnostic output rich enough for the judge step to act on, though the loop prompts were designed around Qwen’s output format and a Nemotron-adapted design remains untested. Nemotron’s loop degrades and even static guidance hurts rather than helps it on this scaffold. The open question is whether the procedure generalizes to a second model that can run it—a larger dense model with strong tool-use behavior, for instance. Scaffolds with context summarization or tree-search may interact with guidance differently regardless of model capacity.

Probe sensitivity. We generate 10 probes per iteration at temperature 0.9 with deduplication and provide a qualitative analysis of what the procedure discovers (Section 3.3), but do not conduct a systematic sensitivity analysis of how the final guidance changes under different probe sets or random seeds. 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 7 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.

11 Conclusion

We introduced probe-and-refine tuning, a lightweight procedure that iteratively refines a statically generated repository knowledge base 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. Across four independent trials on SWE-bench Verified, the resulting guidance achieves a mean resolve rate of 33.0% (mixed-effects logistic regression $p < 0.001$ vs. both baseline and static KB) and produces 31 instances solved consistently (≥ 3 of 4 trials) that no other condition resolves. 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. A qualitative analysis suggests the procedure functions as a localization oracle, helping the agent reach the correct file for small mechanical fixes, rather than enabling fundamentally harder repairs.

The procedure is simple enough to be surprising. That a procedure composed entirely of single-shot LLM calls (~ 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.

A cross-model analysis with Nemotron reveals a model-fit dependency: the procedure appears to require a model that can produce rich diagnostic output and judge its own patch attempts with enough specificity to drive the feedback loop. Below this threshold the loop degrades, guidance interferes rather than helps, and the precision-constancy finding—that per-patch quality is unaffected by guidance—still holds, now replicated under adversarial cross-model transfer.

For practitioners: invest in iteratively refined guidance for repositories where your agent will handle many issues, ensure the step budget accommodates the prescribed workflow, and tune guidance with the model that will consume it, as guidance calibrated for a different model’s behavioral profile may actively harm performance.

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.
- Douglas Bates, Martin Mächler, Ben Bolker, and Steve Walker. Fitting Linear Mixed-Effects Models Using lme4. *Journal of Statistical Software*, 67(1):1–48, 2015.
- 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 Münder, 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 Antoniadis, 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.
- NVIDIA. NVIDIA-Nemotron-3-Nano-30B-A3B. 2026. <https://huggingface.co/nvidia/NVIDIA-Nemotron-3-Nano-30B-A3B-BF16>.