ITERATIVE REPOSITORY GUIDANCE GENERATION FOR CODING AGENTS: EFFECTIVENESS DEPENDS ON STEP BUDGET

A PREPRINT

Asa Shepard
Williams College
as66@williams.edu

March 16, 2026

ABSTRACT

We introduce an iterative oracle-tuning procedure that refines repository-level guidance for AI coding agents through synthetic probes and failure diagnosis, producing repo-specific `AGENTS.md`-style artifacts without access to evaluation data. Evaluating this approach alongside a static knowledge base and an unguided baseline across three step budgets (25, 50, 100) on SWE-bench Verified (500 instances, Qwen3.5-35B-A3B), we find that the agent's step budget—the number of interactive turns before the agent must commit to a patch—moderates whether guidance helps or appears neutral. At 25 steps, all conditions perform equivalently ($\sim$22–24 %). Each guided condition improves at a condition-specific *transition point*: static guidance from 25 to 50 steps ($p = 0.004$), oracle-tuned guidance from 50 to 100 ($p = 0.009$). The unguided baseline remains flat ($\sim$24 %) regardless of budget. At 100 steps, both guided conditions significantly outperform no context ($p < 0.05$), with oracle-tuned guidance achieving the highest point estimate (30.8 %) and the most unique solves, though the gap between the two guided conditions is small (1.2 pp) and not statistically significant. The mechanism is evaluation coverage, not patch quality: precision among evaluated patches is $\sim$56–60 % across all nine cells, but guided agents produce evaluable patches for more instances. These findings suggest one explanation for conflicting prior results on repository context: its effect may depend on whether the agent has sufficient budget to execute the workflow the context prescribes.

1 Introduction

Engineers working with AI coding agents increasingly maintain `AGENTS.md` files documenting repository conventions, entry points, and debugging workflows. Two recent studies reach opposing conclusions about their value. 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 ($\leq$100 lines, $\leq$5 files), measuring efficiency rather than correctness. Gloaguen et al. (2026) find that LLM-generated context files reduce resolve rates by $\sim$3 % on SWE-bench Lite and AGENTbench, with agents following instructions literally even when counterproductive.

We present two contributions that address this disagreement. First, we introduce an *oracle-tuning* procedure that iteratively refines repository-level guidance through synthetic probes and failure diagnosis, producing repo-specific artifacts that outperform both single-pass generated context and no context. Second, we show that the agent's *step budget*—the number of interactive turns before the agent must commit to a patch—moderates the effect of any repository context. Guidance prescribes methodical workflows (reproduce, trace, patch) that consume steps; when the budget is sufficient this methodology leads to better-informed attempts, but when the budget is tight, the agent exhausts its steps on prescribed exploration without reaching the patching phase.

We evaluate three context conditions at three step budgets (25, 50, 100) on 500 SWE-bench Verified instances (SWE-bench, 2024) using Qwen3.5-35B-A3B (Qwen Team, 2025):

1. `no_context`: the bare agent prompt with no repository-specific guidance.

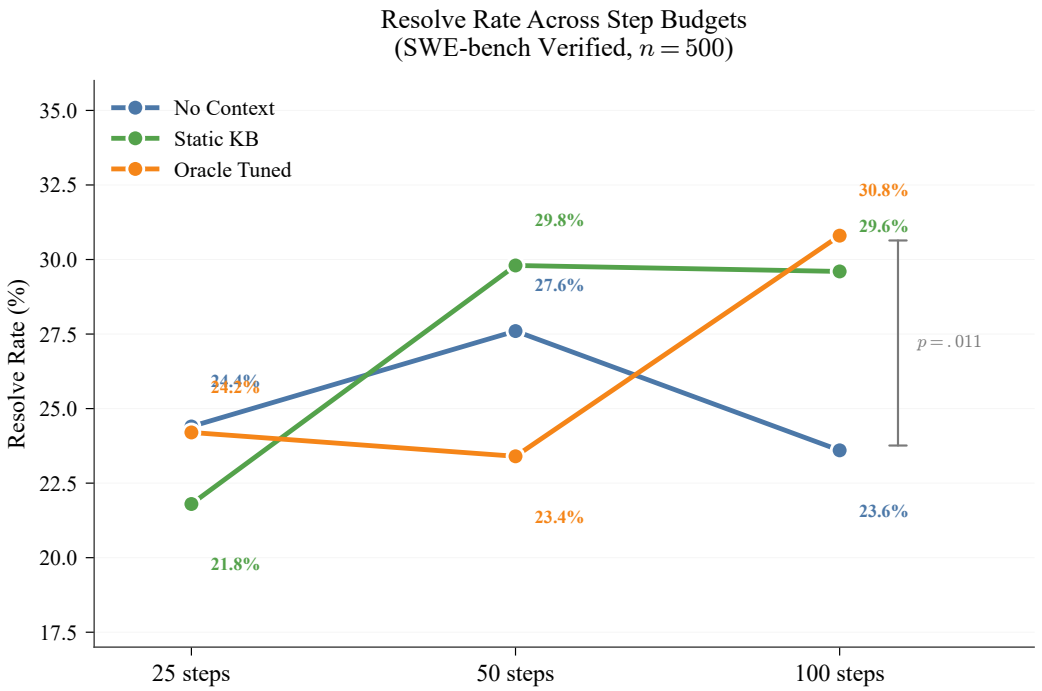

Figure 1: Resolve rate across step budgets on SWE-bench Verified (500 instances) with 95 % Wilson confidence intervals. At 25 steps, all conditions are statistically indistinguishable (∼22–24 %). Guided conditions improve at condition-specific budget transitions, while the unguided baseline remains flat. At 100 steps, both guided conditions significantly outperform no context (oracle-tuned: $p = 0.011$; static KB: $p = 0.032$; two-proportion z-tests).

2. `static_kb`: a fixed per-repo knowledge base combining repository-specific structural analysis (file layout, entry points, import graphs, extracted via tree-sitter) with generic LLM-generated procedural recommendations. This design ensures that `static_kb` already provides both structural and procedural guidance, so that any gains from `oracle_tuned` are attributable to iterative refinement (Section 3.3).
3. `oracle_tuned`: the static KB iteratively refined into repo-specific guidance through 2–5 rounds of synthetic probes, failure diagnosis, and targeted editing.

The results (Figure 1) show that all conditions are equivalent at 25 steps (∼22–24 %), that each guided condition shows a significant improvement at a specific budget transition (static KB from 25 to 50 steps, oracle-tuned from 50 to 100), and that the unguided baseline resolves ∼24 % regardless of budget. Conditional patch precision is ∼56–60 % across all nine cells; the conditions differ not in patch quality but in how many patches they produce.

The paper makes three contributions:

1. An iterative oracle-tuning procedure for repository-level guidance that, at sufficient step budgets, achieves the highest point estimate (30.8 %) and the most unique solves (29) in our study, suggesting that iterative refinement through failure feedback can produce qualitatively different guidance than single-pass generation.
2. A nine-cell experimental design (3 conditions × 3 budgets) establishing that step budget moderates the effect of repository context, with each guidance type showing significant improvement at a different budget transition.
3. A mechanistic analysis showing that the difference is evaluation coverage. Guided agents produce evaluable patches for more instances, while per-patch precision is constant, with trajectory analysis confirming that the unguided baseline's budget invariance reflects exploration-path variance rather than systematic degradation.

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 mentioned tools are used 160× more frequently. Neither study reports or controls

for step budget. Chatlatanagulchai et al. (2025) survey 2,303 context files, finding developers prioritize build commands, implementation details, and architecture. Vasilopoulos (2026) argues single-file manifests do not scale beyond ∼100k lines.

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 simpler pipelines (Xia et al., 2024) and MCTS-based approaches (Antoniades et al., 2025). Published results typically report at a single budget configuration; we hold the scaffold fixed and vary both budget and context.

Repository-level knowledge and self-refinement. RepoGraph (Ouyang et al., 2025) injects code-entity graphs; AutoCodeRover (Zhang et al., 2024) uses AST representations. These provide structured programmatic knowledge, while we study natural-language guidance. Self-Refine (Madaan et al., 2023) and Reflexion (Shinn et al., 2023) iterate on task-level outputs; SWE-ContextBench (Zhu et al., 2026) finds that curated experience helps while unfiltered experience does not. Our oracle tuning operates at the repository level, refining a persistent guidance artifact across many issues as a form of repo-level inference-time adaptation rather than per-task refinement.

3 Method

3.1 System Overview

The system has four stages: (1) repository analysis and context construction; (2) oracle tuning for the `oracle_tuned` condition; (3) patch generation through an interactive agent loop with single-shot fallback; and (4) evaluation via the official SWE-bench Verified harness. All experiments use Qwen/Qwen3.5-35B-A3B (Qwen Team, 2025) with an effective 16k-token context window, 2048 max output tokens per turn, and 512 max tokens for fallback generation. Command outputs are truncated to ∼3000 characters. These constraints—particularly the 16k context window and aggressive output truncation—result in absolute resolve rates well below what larger-context configurations achieve with the same model family; the contribution of this study is the relative comparison across conditions and budgets, not the absolute numbers.

3.2 Context Conditions

`no_context`**.** The baseline 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 such as 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 potential confounds: the structural layer ensures that both `static_kb` and `oracle_tuned` share the same repo-specific foundation, while the procedural layer ensures that any gains from `oracle_tuned` are attributable to iterative refinement rather than the mere presence of generic best-practice guidance.

`oracle_tuned`**.** Starting from `static_kb`, an iterative refinement procedure produces specialized repo-specific guidance over up to 5 iterations. We describe this procedure in detail in Section 3.3. A shared 3000-character guidance cap ensures differences between `static_kb` and `oracle_tuned` reflect content and refinement strategy, not prompt length.

3.3 The Oracle-Tuning Procedure

The oracle-tuning procedure (Figure 2) transforms generic structural knowledge into repo-specific operational guidance through iterative failure feedback. All phases use the same model (Qwen3.5-35B-A3B) as patch generation, ensuring the guidance is optimized for the model that will consume it. Each of 2–5 iterations proceeds in four phases, corresponding to the four stages in Figure 2:

1. **Generate AGENTS.md.** On the first iteration, the system produces the initial guidance artifact from repository structure (this is the `static_kb`). On subsequent iterations, this step is replaced by the output of step 4.

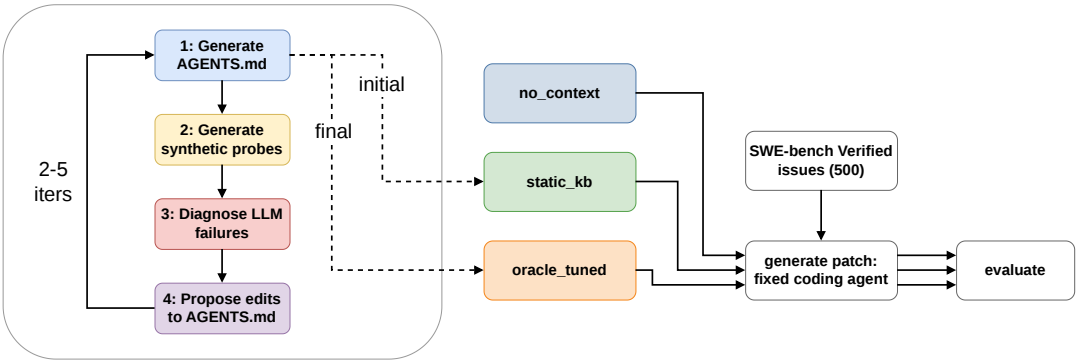

Figure 2: Oracle-tuning pipeline, designed to automate the emerging practice of asking an agent to revise its own `AGENTS.md` after observing failures. The `static_kb` artifact ("initial") feeds both the `static_kb` condition directly and the oracle-tuning loop, which iteratively refines it into the `oracle_tuned` artifact ("final") using synthetic probes, and never using SWE-bench evaluation instances. All three conditions feed the same fixed coding agent for patch generation.

2. **Generate synthetic probes.** The system generates synthetic bug-fix tasks from the repository's codebase (at temperature 0.7), targeting diverse subsystems and failure modes. These probes are distinct from SWE-bench evaluation instances, preventing benchmark leakage.
3. **Diagnose LLM failures.** The agent attempts each probe under the current guidance. Failed attempts are analyzed to identify guidance shortcomings: missing subsystem-specific knowledge, incorrect procedural assumptions, or underspecified debugging strategies.
4. **Propose edits to AGENTS.md.** The guidance artifact is revised to address diagnosed failures, subject to the 3000-character cap. Edits are accepted only if they improve synthetic probe performance on the next iteration.

The result is a qualitative shift from generic to specific. For Django, generic rules like "Verify repo priors with targeted reads" become "Trace through `subclasses.py` to map parent-child relationships" (Figure 3). A meta-pattern emerges independently across all 12 repositories: the oracle converges on a "reproduce first, read second, patch third" workflow, with repo-specific strategies filling in the details of each phase.

This procedure can be understood as *repo-level inference-time adaptation*: rather than refining outputs for a single task (as in Self-Refine (Madaan et al., 2023) or Reflexion (Shinn et al., 2023)), it refines a persistent artifact that shapes agent behavior across all tasks in a repository. The cost is modest: 2–5 iterations of a small number of synthetic probes per repository, with the artifact reusable across all future issues. Importantly, both `static_kb` and `oracle_tuned` guidance artifacts were generated once per repository and then held fixed across all three step-budget conditions, so that differences across budgets reflect only the budget itself, not guidance variation.

3.4 Agent Loop and Fallback Mechanism

The 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. If the agent fails to produce a patch within the step budget, a **single-shot fallback** generates a patch from the issue description and accumulated context.

This fallback is important for interpretation. At low budgets, most instances reach the fallback, making all conditions converge toward fallback quality. At high budgets, most instances are resolved through the agent loop itself, where guidance quality matters. The budget transitions we observe (Section 4.1) reflect, at least in part, the budgets at which each condition's agent loop begins to dominate over the fallback. Patches from both sources are sanitized (removing test-file modifications, enforcing diff format) before evaluation.

3.5 Experimental Design

We evaluate all three conditions at 25, 50, and 100 step budgets on 500 SWE-bench Verified instances, yielding nine experimental cells. The same oracle-tuned guidance artifacts are used at all three budgets. All contexts are generated from pinned commits (Table 1). Temperature is 0.0 for all inference calls except probe generation (0.7). Each cell is a

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 oracle iterations. The oracle independently converges on a reproduce-first workflow with subsystem-specific tracing instructions.

single run; we do not perform repeated runs, a limitation we address in Section 8. All hypothesis tests are two-proportion z-tests (pooled); confidence intervals are 95 % Wilson score intervals.

4 Results

4.1 The Scaling Curve

Table 2 presents resolve rates across all nine cells; Figure 1 visualizes the scaling curves.

Three patterns emerge.

Equivalence at low budgets. At 25 steps, all conditions resolve 21.8–24.4 % with no significant pairwise differences (all $p > 0.3$). At this budget, 44–68 % of instances exhaust their steps and fall through to the single-shot fallback (Section 4.2), so performance reflects fallback quality more than agent quality, and all conditions converge.

Budget-dependent improvement. Each guided condition improves significantly at a specific budget transition. `static_kb` improves from 21.8 % to 29.8 % between 25 and 50 steps ($p = 0.004$), then plateaus. `oracle_tuned` shows no improvement from 25 to 50 steps ($p = 0.766$), then jumps from 23.4 % to 30.8 % between 50 and 100 ($p = 0.009$). One explanation is that static KB provides lightweight structural context consumable in a few steps, while

Table 1: Pinned repository commits. Django accounts for 46 % of instances.

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

Table 2: Resolve rates across step budgets. **Cross-budget** p-values test whether a condition improves between adjacent budgets. **Cross-condition** p-values test guided vs. unguided at each budget. All tests are two-proportion z-tests.

Condition	25 steps	50 steps	100 steps
`no_context`	**24.4 %**	27.6 %	23.6 %
`static_kb`	21.8 %	**29.8 %**	29.6 %
`oracle_tuned`	24.2 %	23.4 %	**30.8 %**
Cross-budget p-values (adjacent transitions):			
`no_context`	25→50: p=.249	50→100: p=.147	
`static_kb`	25→50: $\mathbf{p{=}.004}$	50→100: p=.945	
`oracle_tuned`	25→50: p=.766	50→100: $\mathbf{p{=}.009}$	
Cross-condition p-values (vs. no_context):			
`static_kb`	p=.329	p=.442	$\mathbf{p{=}.032}$
`oracle_tuned`	p=.941	p=.128	$\mathbf{p{=}.011}$

oracle-tuned guidance prescribes a multi-phase workflow (reproduce, trace, patch, verify) that requires more steps to execute. At 50 steps, 38.4 % of oracle-tuned instances still exhaust the budget; at 100 steps, only 14.2 % do.

Budget-invariant baseline. `no_context` shows no significant change at any transition: 24.4 → 27.6 % ($p = 0.249$), 27.6 → 23.6 % ($p = 0.147$), and 24.4 → 23.6 % overall ($p = 0.767$). The unguided agent patches early with 58 % of its patches arriving by step 20 at 100 steps (Section 4.4), and it gains nothing from additional budget.

4.2 Step-Budget Exhaustion and the Fallback

Table 3 shows that guided conditions push more instances to the step ceiling, especially at low budgets.

Table 3: Resource consumption across all nine cells. **Exhaustion**: instances reaching the step ceiling. **Fallback**: instances where the single-shot fallback generated the final patch. **Mean steps**: average steps per instance.

Budget	Condition	Exhaustion	Fallback	Mean steps	Prompt tok.
25	`no_context`	44.2%	28.8%	18.9	57M
	`static_kb`	62.0%	50.0%	21.3	69M
	`oracle_tuned`	67.6%	23.2%	22.4	75M
50	`no_context`	20.8%	19.8%	27.4	86M
	`static_kb`	33.2%	31.4%	32.5	109M
	`oracle_tuned`	38.4%	24.4%	34.7	121M
100	`no_context`	9.2%	24.6%	33.2	105M
	`static_kb`	10.0%	34.2%	41.0	140M
	`oracle_tuned`	14.2%	15.2%	46.4	166M

At 100 steps, oracle-tuned has the lowest fallback rate (15.2 %, vs. 24.6 % for no-context and 34.2 % for static-KB), meaning it most often produces patches through its own agent loop, the regime where its structured workflow pays off. The 14.2 % exhaustion rate suggests room for further improvement at higher budgets.

Mean steps increase with budget for all conditions, but this additional exploration translates to higher resolve rates only for guided conditions. The unguided baseline uses 76 % more steps at 100 than at 25—consuming 105M prompt tokens vs. 57M—with no resolve-rate gain, meaning nearly half its total compute is spent on exploration that produces no additional resolved instances.

4.3 The Mechanism: Coverage, Not Patch Quality

Conditional patch precision (resolved / evaluated) is stable across conditions and budgets (Table 4), falling within 55.4–60.0 % at 50 and 100 steps with no significant pairwise differences (all $p > 0.5$).

Table 4: Precision (resolved / evaluated) with 95 % Wilson CIs. No pairwise comparison reaches significance (all $p > 0.5$).

Condition	50 steps	100 steps
`no_context`	55.9 % [49.6, 61.9]	58.7 % [51.8, 65.3]
`static_kb`	55.4 % [49.4, 61.2]	59.7 % [53.5, 65.6]
`oracle_tuned`	60.0 % [53.0, 66.6]	56.8 % [50.9, 62.6]

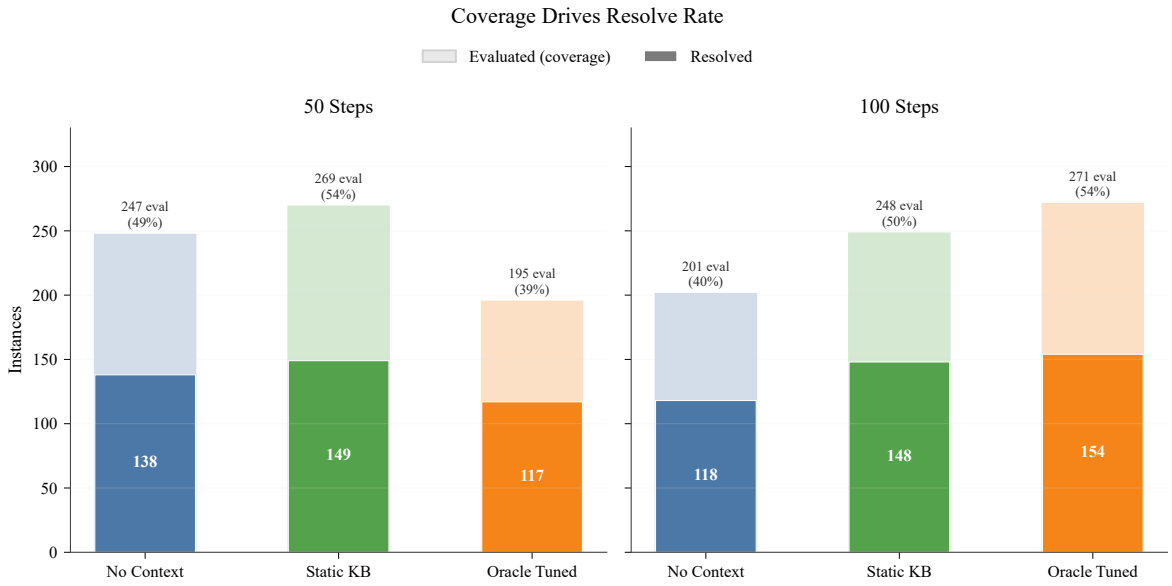

Figure 4: Coverage drives resolve rate. At 50 steps, oracle-tuned has the lowest coverage (39.0 %); at 100 steps, the highest (54.2 %). Precision is similar throughout (∼56–60 %).

The resolve-rate differences are driven entirely by evaluation coverage: the fraction of instances that produce evaluable patches (Figure 4). At 50 steps, oracle-tuned evaluates only 195/500 instances (39.0 %) while static-KB evaluates 269 (53.8 %). At 100 steps, oracle-tuned evaluates 271 (54.2 %), the most of any condition, while no-context evaluates only 201 (40.2 %). This coverage shift drives the ranking reversal.

What does guidance actually change about agent behavior? The step-distribution data (Section 4.4) offers a concrete picture. Under `no_context`, the agent's early steps are also its patching steps: it reads the issue, explores briefly, and commits to a patch by step 20 in the majority of cases. Under `oracle_tuned`, the early steps are devoted to the prescribed reproduce-and-trace workflow: running tests, reading specific files, following call chains. This workflow builds context that informs a later patch but does not itself constitute one. These early steps are productive (they accumulate diagnostic information the agent uses later) but would be wasted under a tight budget, because the agent never reaches the patching phase. This explains both why oracle-tuned guidance appears neutral or harmful at low

budgets and why it achieves the highest resolve rate when the budget is sufficient: the guidance front-loads diagnostic work that pays off only if the agent has enough remaining steps to act on what it learned.

Guidance does not make individual patches better (as observed, per-patch precision is constant) but it restructures *when* and *how* the agent commits to a patch, trading early impulsive attempts for later informed ones.

4.4 Late-Step Productivity

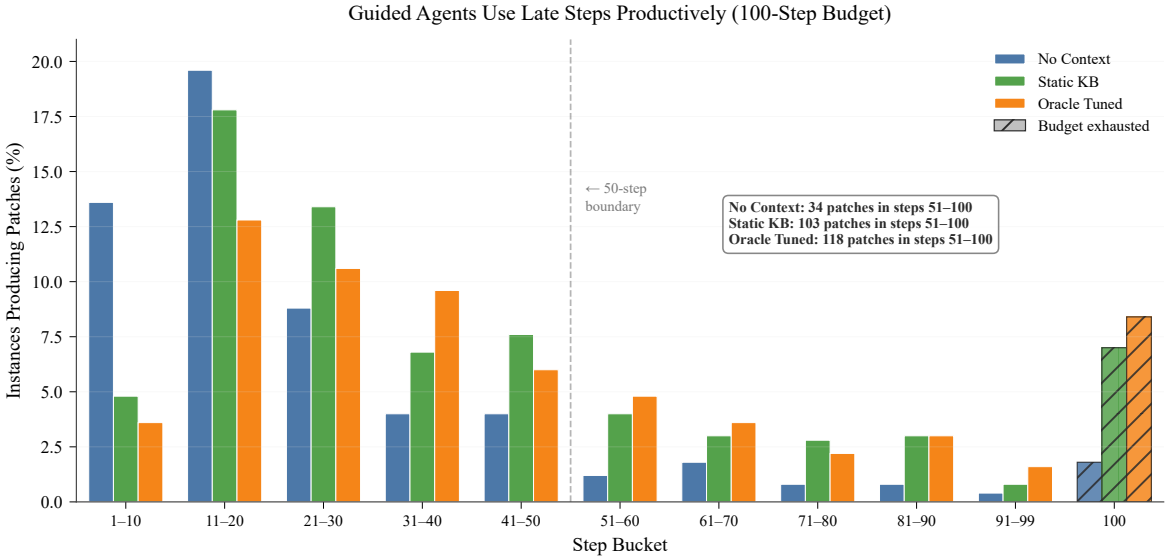

Figure 5: Step distribution of patches at 100 steps. `no_context` is front-loaded (58 % by step 20), producing only 34 patches in steps 51–100. Guided agents produce 103–118 late-step patches.

The step distribution at 100 steps (Figure 5) reveals the mechanism behind budget invariance. `no_context` produces 58.5 % of its patches in the first 20 steps and only 34 from steps 51–100. `oracle_tuned` produces 118 patches from steps 51–100; `static_kb` produces 103. The unguided agent patches early or not at all; the guided agents use late steps for verification and refinement phases of their prescribed workflows.

4.5 Complementarity and Per-Repository Variation

Each condition uniquely solves instances the others cannot (Table 5). At 100 steps, oracle-tuned produces the most unique solves (29), and the union of all three conditions resolves 207 instances (41.4 %), 34 % more than the best single condition.

Table 5: Unique solves across budgets.

Condition	Unique (50 steps)	Unique (100 steps)	Δ
`oracle_tuned`	20	**29**	+9
`no_context`	25	21	−4
`static_kb`	26	16	−10
Union (all 3)	206 (41.2%)	207 (41.4%)	+1

Per-repository variation is substantial but based on small subgroup sizes. On matplotlib, oracle-tuned achieves 76.9 % precision vs. 38.5 % for the baseline—the largest gap in the study—using guidance with specific instructions for 3D rendering paths and DPI-related test reproduction. On xarray, both guided conditions outperform no-context by ∼37 pp. On sympy, all conditions converge at 50.0 %. Because Django accounts for 46 % of instances, we verified that the core pattern—guided conditions outperforming no-context at 100 steps, with a flat unguided baseline—holds on both the Django subset and the non-Django subset, though subgroup sizes limit statistical power for the latter. This heterogeneity motivates adaptive context selection (Li et al., 2026; Zhu et al., 2026) where guidance depth is matched to repository characteristics.

5 Trajectory Analysis

The unguided baseline resolves 27.6 % at 50 steps and 23.6 % at 100—a 4.0 pp drop that does not reach significance ($p = 0.147$). To understand this, we analyzed the 100-step trajectories of 43 instances that `no_context` resolved at 50 steps but failed at 100.

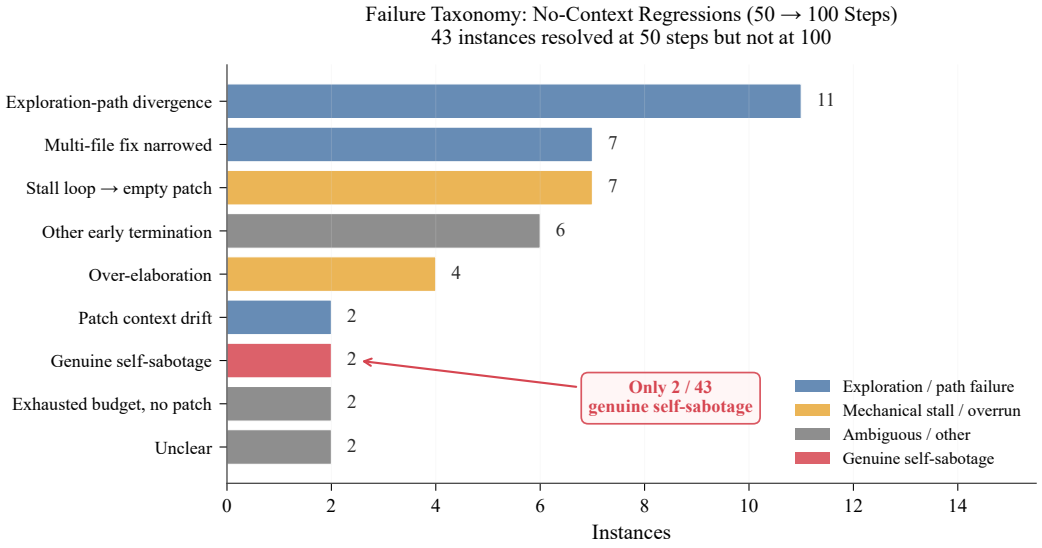

Figure 6: Failure taxonomy of 43 regression instances (resolved at 50 steps but not at 100), color-coded by failure family. Exploration and path-related failures (blue) dominate; mechanical stalls and over-elaboration (amber) are secondary. Genuine self-sabotage (red) accounts for only 2 of 43 cases.

The failures cluster into three families (Figure 6). The largest is **exploration/path failures** (18 of 43 cases): the 50-step and 100-step runs are independent executions that explore the codebase along different trajectories, and on these instances the 50-step run happened to find better paths. This includes 11 cases of semantically related but incorrect fixes and 7 cases where a correct multi-file fix was narrowed to a single file. Crucially, 17 of 30 wrong non-empty patches were produced before step 15, the same step range where the 50-step run also patches, ruling out additional steps as the cause of failure.

Mechanical stalls (11 cases) include stall loops where the agent enters a repeated-command cycle (7 cases) and over-elaboration where the agent finds a correct approach but continues iterating until it degrades its own solution (4 cases).

Genuine self-sabotage, the agent producing a correct patch and then progressively stripping it, accounts for only **2 of 43 cases**. The 4.0 pp decline is run-to-run variance, not a systematic failure mode: the unguided agent does not harm itself with more steps, but it does not benefit from them either, because it lacks a structured plan for how to use additional budget.

6 Relating to Prior Findings

Our results suggest one possible explanation for the disagreement between Lulla et al. (2026) and Gloaguen et al. (2026), though important caveats apply.

Neither study reports or controls for step budget. Lulla et al. (2026) evaluate focused pull requests where agent budgets are likely generous relative to task complexity; Gloaguen et al. (2026) evaluate more complex benchmark tasks under default settings. Our results show that the same guidance artifact can appear neutral, harmful, or beneficial depending on budget. If the effective budgets in the two studies differ relative to task complexity, this could contribute to their opposing conclusions, though the studies also differ in agents, models, and benchmarks.

Two specific observations are relevant. First, Gloaguen et al. (2026) frame literal compliance with context files as counterproductive; our results suggest it is counterproductive only when the budget cannot accommodate the prescribed workflow. Second, Gloaguen et al. (2026)'s context files are generated in a single LLM pass, while our oracle-tuned

guidance is iteratively refined through failure feedback. At 100 steps, oracle-tuned achieves the highest resolve rate and the most unique solves, consistent with the finding of Zhu et al. (2026) that curated experience helps while unfiltered experience does not.

7 Discussion

Practical implications. At 100 steps, both guided conditions significantly outperform the baseline ($p = 0.011$ and $p = 0.032$). Even a static KB, constructable automatically from repository structure, is sufficient to improve resolve rate over the baseline at budgets $\geq$50. The token-cost tradeoff is worth stating explicitly: oracle-tuned guidance at 100 steps consumes 166M prompt tokens to achieve 30.8 % resolve rate, while no-context at 100 steps consumes 105M tokens for only 23.6 %. The guided condition spends 58 % more tokens but resolves 30 % more instances, and this gap would likely widen at higher budgets given oracle-tuned's 14.2 % exhaustion rate. For frontier models with stronger instruction-following, we hypothesize that budget transition points would be lower: the agent would execute guided workflows in fewer steps, making richer guidance practical even at moderate budgets. More broadly, scaling compute without scaling guidance appears ineffective: the unguided baseline resolves $\sim$24 % regardless of budget, suggesting that step budget and guidance quality are complementary dimensions that must be scaled together.

Cost-effectiveness of static KB. A cost-conscious practitioner should note that `static_kb` at 50 steps achieves 29.8 % with 109M prompt tokens, nearly matching oracle-tuned's 30.8 % at 100 steps (166M tokens)—and at 100 steps the two guided conditions are statistically indistinguishable ($p > 0.5$). The static KB requires no iterative refinement and can be constructed automatically from repository structure via tree-sitter in minutes. For compute-constrained settings, static guidance with moderate budgets may offer the best cost–performance tradeoff. Oracle tuning is most justified when practitioners already operate at high step budgets and seek marginal gains, or when per-repository investment can be amortized across many issues.

Interpreting absolute performance. The resolve rates (22–31 %) reflect a constrained pipeline: a 35B MoE model with 16k context and aggressive truncation. These are not comparable to frontier results (>50 %); the contribution is the relative comparison within a fixed pipeline.

Oracle tuning as repo-level adaptation. The oracle-tuning procedure occupies a middle ground between per-task self-refinement and static documentation. It requires modest compute (2–5 iterations of synthetic probes per repository), produces reusable artifacts, and at sufficient budgets achieves the highest point estimate in our study. The 14.2 % exhaustion rate at 100 steps suggests that even higher budgets would further benefit oracle-tuned guidance. The oracle-tuned condition receives more total compute investment than static KB during guidance construction. However, this asymmetry is inherent to the method being proposed, not a confound: the static KB's structural layer is deterministic (tree-sitter extraction), and its procedural layer is deliberately generic. The relevant question is whether the iterative refinement procedure justifies its cost, and the answer depends on the deployment context (see cost-effectiveness discussion above). The 1.2 pp gap between oracle-tuned and static KB at 100 steps is not statistically significant in a single run; the stronger claim is that both guided conditions outperform the unguided baseline, and that oracle-tuned produces the most unique solves (29 vs. 16 for static KB), suggesting qualitatively different coverage even if aggregate rates are similar.

8 Limitations

Single model and scaffold. Results are from one model (Qwen3.5-35B-A3B) with one custom pipeline. Frontier models may show budget-dependent improvement at lower step counts, and scaffolds with summarization or memory may alter the budget–guidance interaction.

No repeated runs. Each cell is a single run. The trajectory analysis suggests $\sim$4 pp of run-to-run variance, meaning the difference between the two guided conditions (1.2 pp at 100 steps) is within noise; only the guided-vs.-unguided comparisons ($\geq$6 pp) are robust to this variance.

Fallback confound. The single-shot fallback means step budget simultaneously affects agent exploration time and fallback trigger point. The budget transitions may partly reflect when each condition's fallback rate drops below a critical level; this mechanism is specific to our scaffold and may not generalize to scaffolds without a fallback.

Limited budget granularity. Three budget points cannot sharpen transition-point estimates or determine whether oracle-tuned continues improving beyond 100 steps.

Single benchmark. Findings are specific to SWE-bench Verified (500 instances).

9 Conclusion

We introduced an iterative oracle-tuning procedure for repository-level guidance and showed that the agent's step budget moderates whether such guidance helps. At 25 steps all conditions are equivalent; as budget increases, both guided conditions improve at distinct budget transitions while the unguided baseline remains flat. At 100 steps, both guided conditions significantly outperform the baseline, with oracle-tuned guidance achieving the highest point estimate (30.8 %, $p = 0.011$ vs. baseline) and the most unique solves (29). The mechanism is evaluation coverage, not patch quality.

For practitioners: provide structured guidance and ensure the step budget accommodates the guidance depth. Iteratively refined, repo-specific guidance requires more steps but achieves the best results when those steps are available.

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. 2025. `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.

Xingyao Wang, Boxuan Li, et al. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. *ICLR*, 2025. arXiv:2407.16741.

Antonis Antoniades, Albert Örwall, Kilian Lieret, Xingyao Wang, and Oleksii Kuchaiev. SWE-Search: Enhancing Software Agents with Monte Carlo Tree Search and Iterative Refinement. *ICLR*, 2025.

Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. 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.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the Middle: How Language Models Use Long Contexts. *TACL*, 2024.

Yufeng Du, Minyang Tian, et al. Context Length Alone Hurts LLM Performance Despite Perfect Retrieval. *Findings of EMNLP*, 2025. arXiv:2510.05381.

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.

Han Li et al. ContextBench: A Benchmark for Context Retrieval in Coding Agents. *arXiv:2602.05892*, 2026.

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

Cognition. SWE-bench Technical Report. 2024. `https://cognition.ai/blog/swe-bench-technical-report`.

Benjamin Wei et al. ASyMOB: Algebraic Symbolic Mathematical Operations Benchmark for LLMs. *arXiv:2505.23851*, 2025.