LEANREPAIR: TOKEN-EFFICIENT EXTERNAL CONTROL FOR LOCAL CODE REPAIR

Anonymous authors
Paper under double-blind review

ABSTRACT

Local coding agents keep source code private and give users direct control over deployment, but multi-candidate test-time reasoning incurs severe token costs that strain local serving budgets: under standard branch-and-merge search, every unguided generation, aggregation, and replayed context token burns inference computation without guaranteeing progress. We introduce **LeanRepair**, a deterministic external controller designed to improve the token efficiency of local code repair while maintaining competitive pass rates, requiring no training and no additional language-model calls. LeanRepair addresses two complementary sources of token expenditure: (1) context inflation, bounded by maintaining compact external edit-geometry state rather than replaying verbose histories, and (2) unproductive search persistence, mitigated via trajectory-aware early stopping that prunes remaining branch expansions when edit trajectories oscillate or stagnate. Across six repair benchmarks and four instruction-tuned backbones (24 cells), LeanRepair reduces token expenditure compared to Graph-of-Thoughts by a median of 23.9% across cells (and 23.0% pooled across the five non-QuixBugs benchmarks, reaching 82.4% on QuixBugs where baseline search suffers pathologically long loops), while achieving a competitive 75.4% aggregate pass rate (vs. 72.9% for GoT; 15 wins, 3 ties, 6 losses). Our findings demonstrate that lightweight, non-parametric external control can substantially reduce test-time token waste in local code repair without relying on expensive model-based reflection or learned value functions.

1 INTRODUCTION

Coding agents have become an important testbed for progress toward general-purpose AI because software development combines planning, tool use, environment interaction, and iterative correction under verifiable feedback (**??**). Many capable agents rely on hosted language models, yet repeatedly sending source code to an external service can raise privacy and governance concerns, and long repair trajectories can incur substantial *token* cost—the primary billable unit of LLM inference. Local coding agents are therefore important for privacy-sensitive and resource-constrained settings: they keep code within the user's environment and provide direct control over deployment, but their smaller models and finite token budgets make inefficient multi-candidate repair especially costly. Under a fixed attempt and branch budget, every unused generation, aggregation, and replayed context token competes directly with the computation available to produce and test a correct fix.

Current systems improve code through several complementary strategies. Self-Refine uses model-generated critique, explanation, and execution feedback to revise a candidate (Madaan et al., 2023). Tree-of-Thoughts and Graph-of-Thoughts organize multiple candidates into structured search (Yao et al., 2023; Besta et al., 2024), while CodeTree uses execution and model-generated feedback to guide ranking, expansion, and termination (Li et al., 2025). Agent frameworks add explicit control and memory: RepairAgent uses a finite-state machine for tool use (Bouzenia et al., 2025), Reflexion stores verbal reflections across trials (Shinn et al., 2023), and bounded-context methods compress accumulated reasoning or observations (Yang et al., 2025; Kang et al., 2025). Adaptive methods can also calibrate when reasoning should stop under a compute budget (Wang et al., 2026).

These solutions introduce severe *token* costs that are especially penalizing for local models. Generating critiques or memory summaries requires additional inference tokens; learned evaluators require training; calibrated stopping requires validation data; and replaying or compressing a growing his-

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

tory increases prompt tokens quadratically with every attempt. Structured search further spends a fixed branch width at each active attempt unless an external mechanism decides that the remaining tokens are unlikely to help. Execution feedback identifies the immediate failure, but does not by itself describe whether the latest revision was nearly unchanged, rewrote too broadly, ignored the feedback, or returned to a program that had already failed. The local repair problem is therefore fundamentally a **token-allocation** control problem: after every unsuccessful attempt, the system must decide how to steer the next revision, what limited history to retain, and whether another branch-and-merge step is worth its token cost.

We are motivated by the observation that model-based self-reflection requires additional inference tokens and may repeatedly condition on earlier unsuccessful reasoning, whereas external trajectory features can be computed deterministically without another model call. We view iterative repair as a trajectory through program space and identify three potential failure motifs: *stagnation* (micro-edits that fail identically), *oscillation* (alternating between conflicting partial patches), and *recurrence* (cycling back to previously rejected programs). Crucially, our ablations uncover an unexpected *oscillation asymmetry*: frozen LLMs under execution feedback rarely produce stagnant micro-edits; instead, test-time token churn is overwhelmingly driven by oscillation, meaning oscillation detection alone captures all stopping token savings. Instead of asking the model to introspect, an external controller can monitor lightweight trajectory statistics externally and map them to either a concise directive or a stop action, avoiding paying another LLM to reason about whether more LLM reasoning is worthwhile. This leads to a concrete question:

> *How effectively can external trajectory monitoring eliminate unproductive search churn in local code repair, cutting token expenditure while preserving or improving pass rates?*

We introduce **LeanRepair** to investigate this question around a fixed branch-and-merge repair backend. LeanRepair's clearest gain is token efficiency: it extracts edit-geometry features from the latest program transition and reuses them for three token-conserving purposes. First, *oscillation-aware early stopping* uses a short window of edit changes to avoid remaining branch expansions when the trajectory oscillates or stagnates—the primary engine for cutting candidate generation tokens. Second, *edit-geometry guidance* maps the same features to a short recovery instruction; edit magnitude, program similarity, feedback overlap, and error type act as heuristic indicators of the revision regime without adding verbal critique tokens. Third, a *finite-horizon recurrence buffer* compares the current failed program with recent rejected candidates and overrides ordinary guidance when a prior program recurs, preventing cyclical token burn.

Our contributions are as follows:

- **Token-lean external control architecture**: We formulate iterative code repair under a non-parametric controller that decouples process steering from candidate synthesis, guaranteeing $\Theta(J)$ prompt complexity versus full-history replay ($\Theta(J^2)$) and providing lightweight search pruning with zero controller token overhead.
- **Empirical discovery of the oscillation asymmetry**: While iterative failure is often conceptualized as stagnation, our ablations reveal a sharp asymmetry: LLM repair trajectories rarely stagnate (saving 0% tokens), but instead suffer from high-variance edit oscillation. Consequently, oscillation-only detection captures 100% of stopping savings at iso-accuracy.
- **Multi-backbone empirical evaluation**: Across six benchmarks and four instruction-tuned model families (24 evaluation cells), LeanRepair reduces tokens in 23 of 24 cells (median 23.9% reduction vs. GoT) while maintaining competitive pass rates (75.4% vs. 72.9%; 15 wins, 3 ties, 6 losses).

2 PRELIMINARIES

Iterative code repair. Given a programming task and a program that fails its tests, a language model proposes a revised program, and an execution environment runs the tests and returns feedback (a syntax error, an incorrect output, or a timeout); if the revised program passes, repair succeeds, otherwise the model can try again with that feedback until its attempt budget is exhausted, a sequence

we call a *repair episode*. A method may generate one candidate program per attempt or explore several branches and combine them into a further candidate for testing.

Programs, feedback, and repair history. Let q denote the programming task, M a fixed pre-trained language model, and J the maximum number of repair attempts; after an unsuccessful attempt j, let c_j be the retained failed program and f_j its execution feedback (in our multi-candidate implementation, the merged candidate and its feedback; Section 3.2). The history of unsuccessful attempts is

$$\mathcal{T}_{\leq j} = \big((c_i, f_i)\big)_{i=1}^{j}.$$

The latest pair describes what failed, while comparisons across attempts describe how the process is evolving—for example, a nearly unchanged program with the same error suggests an ineffective revision, and a return to an earlier failed program suggests repetition—but retaining every previous program and traceback also makes the prompt context supplied to the model grow with the number of attempts.

Inference token expenditure. The computational cost of a repair episode is governed by its total token spend: input (prompt) tokens and output (generation) tokens. In multi-candidate search with branch width b over J attempts, naive agent loops incur two compounding token penalties: (i) *Prompt inflation*: replaying full histories $\mathcal{T}_{\leq j}$ compounds input length on every attempt; (ii) *Unproductive branch expansion*: fixed-budget search expends b branch completions plus an aggregation call at every attempt $j \leq J$, even when the model is trapped in repetitive or stagnant failures.

3 LeanRepair Design

3.1 Problem Formulation and Token Economics

Control problem and trajectory failure motifs. We view iterative repair as a trajectory through program space, where successive transitions $c_j \rightarrow c_{j+1}$ are generated by sampling model M across branch width b up to attempt budget J. Unproductive repair episodes are characterized by three recurring failure motifs: (i) *Edit stagnation*, where edit distances collapse below a threshold ($\|\Delta c_j\| < \tau_{\text{stag}}$) as the generator repeatedly outputs near-identical failing micro-edits; (ii) *Edit oscillation*, where second-order edit differences $\Delta^2 c_j$ alternate in sign as the model flips between conflicting partial fixes; and (iii) *Program recurrence*, where the search revisits previously rejected code states ($c_j \in \mathcal{C}_{\leq j}$). While existing agent frameworks rely on expensive model-based self-critiques, LeanRepair introduces an external controller that emits a continuation bit $u_j \in \{0, 1\}$ and recovery directive m_{j+1} from lightweight edit-geometry statistics, instantiating three token-conserving operators:

1. **Trajectory-aware early stopping.** A stop rule $u_j = u(\mathcal{D}_j, j)$ driven by a short path of edit-geometry features; setting $u_j{=}0$ at attempt j^* avoids up to $b(J{-}j^*)$ remaining candidate generations, together with their aggregation calls—the primary engine of token reduction (Section 3.5).
2. **Edit-geometry guidance.** A map h such that $m_{j+1} = h(a_j)$ depends on the latest edge feature $a_j = \phi(c_{j-1}, c_j, f_{j-1}, f_j)$; this provides targeted steering via short directives ($|m_{j+1}| \leq 15$ tokens) rather than expensive multi-token model critiques (Section 3.3).
3. **Finite-horizon recurrence buffer.** A window $\mathcal{C}_j$ of the last K rejected programs; if c_j matches an earlier entry, h is overridden by a recurrence instruction, preventing cyclic token waste without expanding the prompt context (Section 3.4).

Proposition 1 (Quadratic-vs-linear episode prompt cost)**.** *For a fixed task q, suppose each candidate program and feedback string has length $\Theta(\ell)$ tokens for constant $\ell > 0$, with branch width b. Over J executed attempts, full-history replay incurs $\Theta(J^2)$ prompt tokens, while a bounded-state controller incurs $\Theta(J)$ prompt tokens (proof in Appendix A.3).*

Proposition 1 formalizes the asymptotic prompt advantage of external state relative to uncompressed conversational agents (e.g., standard multi-turn refinement or Reflexion) that accumulate history transcripts. When comparing against structured-search backends like GoT and ToT, which already maintain bounded prompts per attempt, the empirical token savings stem from a distinct mechanism:

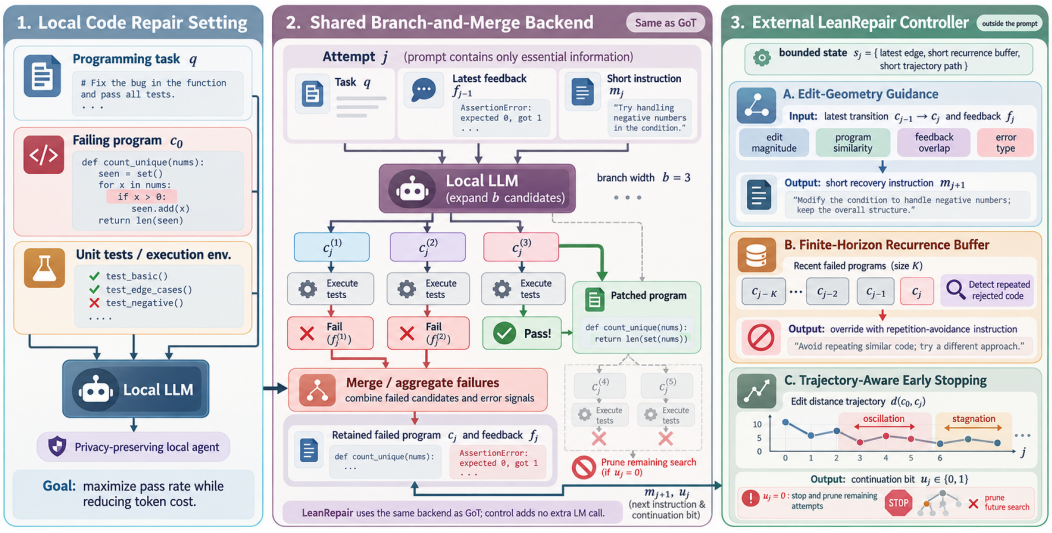


Figure 1: **LeanRepair controller and search backend.** From a bounded external state the controller emits a continuation bit u_j and, if $u_j = 1$, a edit-geometry recovery instruction m_{j+1}. Guidance maps the latest program edge to m_{j+1}; the recurrence buffer can override that instruction; u_j stops the remaining branch-and-merge computation when the trajectory trigger fires (Sections 3.3–3.5). Computing the next edge requires retaining the current program and feedback outside the model; the feature vector alone is not the complete operational memory.

trajectory-aware early stopping, which prunes unpromising candidate expansions on the output generation path. Coupled together, external state and early stopping provide token containment across both prompt inputs and branch outputs. The remainder of this section develops each piece.

3.2 Framework Overview

LeanRepair is an external controller on a matched search backend of width b and depth budget J, answering three questions the generator is poorly positioned to answer alone: in what regime should the next edit be made (edit-geometry guidance), has this candidate already failed within a recent horizon (recurrence buffer), and is the remaining branch-and-merge computation worth its cost (early stopping)? After a failed attempt $j < J$ it emits two outputs from a retained state s_j:

$$u_j = u(s_j, j) \in \{0, 1\}, \qquad m_{j+1} = h(s_j) \text{ if } u_j = 1. \tag{1}$$

If u_j=0, early stopping terminates the episode and avoids the remaining attempts, candidate generations, and aggregation calls; otherwise guidance supplies a short recovery instruction m_{j+1} and the backend generates $c_{j+1} \sim B(q, f_j, m_{j+1})$. The language model never sees s_j: the prompt contains only the task, the latest feedback, and m_{j+1}, so the controller retains process memory while the model receives just the consequence needed for the next synthesis step.

The pieces of s_j are developed in Sections 3.3–3.5; the main evaluation measures their combined effect rather than isolating each one. The operational state is

$$s_j = (a_j, \mathcal{C}_j, f_j, \mathcal{D}_j),$$

where a_j is the latest-edge (action) feature, $\mathcal{C}_j$ is the finite-horizon recurrence buffer of recent programs including c_j, and $\mathcal{D}_j$ retains the guidance-feature path used for pruning; a new candidate is compared with the retained programs before the oldest entry is discarded. Algorithm 1 states the headline procedure, with numerical constants confined to Appendix A.1.

Single-lag LeanRepair implements only guidance; buffered LeanRepair adds the recurrence window; **buffered LeanRepair with trajectory-aware early stopping** adds u_j and is the default **LeanRepair** in Table 1. The precision studies keep $u_j = 1$ for all $j < J$; the scale curve uses the same pruning controller as Table 1.

The main experiments use the same branch-and-merge backend as GoT (Besta et al., 2024): at an active attempt j it generates a fixed number of candidates,

$$\{c_j^{(1)}, \ldots, c_j^{(b)}\} \sim M(q, f_{j-1}, m_j),$$

executes them, returns if any pass, and otherwise merges the failing branches into one aggregated program that the controller updates from if it also fails. GoT and LeanRepair therefore issue the same b generation calls and at most one aggregation call per active attempt, and LeanRepair adds no model call for control; removing the recovery instruction and forcing $u_j = 1$ recovers plain GoT (removing the instruction alone does not, since GoT has no pruning rule), so Table 1 compares a combined change in guidance and pruning under a shared per-attempt search structure. A single-candidate backend (unit branch width, no aggregation) is also compatible with the same controller (Appendix A.2).

3.3 Edit-Geometry Guidance

Guidance answers which *regime* the next revision should occupy: the map h classifies the latest program edge rather than reconstructing a global picture of the episode. The same execution failure can be followed by a near-unchanged program, a broad rewrite, or a revision that does not incorporate the feedback; the transition between c_{j-1} and c_j helps distinguish these cases.

Single-lag LeanRepair forms an action feature from consecutive failed programs and feedback,

$$a_j = \phi(c_{j-1}, c_j, f_{j-1}, f_j),$$

and sets $m_{j+1} = h(a_j)$ (or $h(s_j)$ once the recurrence buffer is active). The coordinates of a_j are edit magnitude, token similarity, lexical overlap between consecutive feedback strings, and the current error type (syntax, assertion, type/value, or timeout). Edit magnitude and similarity detect *collapse* (a near-zero edit that leaves the failure unchanged) and *explosion* (a broad rewrite that does not use the traceback). Feedback-term overlap is a lexical proxy for incorporation, not a semantic guarantee that the failure was addressed.

Fixed rules with a deterministic priority among collapse, explosion, unused feedback, and error type map these features to a short instruction (Appendix A.1): collapse requests a substantive edit, explosion with weak overlap requests a targeted correction, and a timeout requests a change of algorithm. The instruction is short because the geometry has already classified the regime—the frozen model generates under a constraint rather than reconstructing the process theory that produced it—and the rules are hypotheses about useful next actions, not a learned policy or an extra model call.

3.4 Finite-Horizon Recurrence Buffer

The recurrence buffer answers whether the current candidate has already failed—a classical closed-list / taboo question that pure single-lag locality cannot ask; recent memory systems likewise treat *recurrence* as the signal that a past interaction warrants special handling. The last-edge map ϕ sees immediate repetition but not a return two or more attempts earlier; locality is blind to cycles that close beyond lag one.

Proposition 2 (Historical ambiguity of single-lag state)**.** *Without restrictions on admissible histories, no state $s_j = \sigma(c_{j-1}, c_j, f_{j-1}, f_j)$ can determine for every history whether c_j occurred before attempt $j-1$. Distinct histories can share the same latest transition while differing on this recurrence event (proof in Appendix A.3).*

A window of size K resolves recurrences inside that horizon and still cannot preserve every constraint from an arbitrarily long repair history.

LeanRepair therefore maintains a finite-horizon buffer $\mathcal{C}_j$ of the last K rejected programs; if c_j's token-Jaccard similarity to an earlier window entry (excluding the immediate predecessor) exceeds a fixed threshold, the buffer overrides guidance with a repetition-avoidance instruction, and only that instruction enters the prompt—a closed-list in program space, not a replayed transcript. This enlarges memory only where the single-lag map is provably incomplete, not by defaulting to full history: the horizon is fixed, so repetitions beyond it stay invisible, and the threshold does not claim every near-duplicate is harmful. Table 6 shows this buffer matters most when guidance alone is harmful (Gemma) and is otherwise a small correction; constants are in Appendix A.1.

3.5 TRAJECTORY-AWARE EARLY STOPPING

Pruning answers whether the residual search tree is still worth expanding. Classical LLM tree search prunes by an external value score—a language-model evaluator in ToT (Yao et al., 2023), a process reward, or an adaptive width/depth policy—which adds calls and separates "stop" from "steer." LeanRepair instead ties pruning to the same guidance geometry that produces m_{j+1}, reading a short path of action features in $\mathcal{D}_j$ rather than scoring nodes with an additional call.

The headline rule inspects unsuccessful trajectories after a fixed number of attempts: over a short trailing window of edit distances it sets $u_j = 0$—discarding every remaining depth $j' > j$ and the b candidates each would have generated—if consecutive differences change sign often enough (oscillation) or mean edit distance is small (stagnation), the saturated regimes of the same collapse/explosion geometry used by h; otherwise $u_j = 1$ until a candidate passes or the budget J is reached. This early-stopping decision is driven by guidance features rather than a confidence score or an extra model call, and does not claim the model cannot solve the task—only that *this* trajectory has entered a regime where further computation is unlikely to be informative.

4 EXPERIMENT

We structure our evaluation around three core investigations: a four-backbone comparison across six benchmarks (Section 4.2), parameter scaling from 0.6B to 8B (Section 4.3), and weight quantization under 4-bit, 8-bit, and 16-bit precisions (Section 4.4).

4.1 EXPERIMENT SETUP

Dataset and Backbones. We evaluate on six benchmarks spanning algorithmic logic, execution reasoning, and repository libraries: HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), CRUXEval (Gu et al., 2024), APPS (Hendrycks et al., 2021), BigCodeBench (Zhuo et al., 2025), and QuixBugs (Lin et al., 2017) (Appendix B.3). Across these tasks, we evaluate four instruction-tuned models: Llama-3-8B-Instruct, Phi-3.5-mini-instruct (3.8B), Gemma-2-9B-It, and Qwen3-8B (Appendix B).

Baselines. We compare against Self-Refine (Madaan et al., 2023), zero-shot CoT (Wei et al., 2022), Tree-of-Thoughts (Yao et al., 2023), and Graph-of-Thoughts (Besta et al., 2024).

Environment and Metrics. Models run locally in PyTorch on an NVIDIA A100 GPU using HuggingFace `transformers`. We measure pass rate (%) to evaluate repair success and tokens per task (prompt and generation tokens) as the primary inference cost metric.

4.2 MAIN RESULTS

We evaluate four instruction-tuned backbones (Llama, Phi, Gemma, and Qwen3) on six repair benchmarks: QuixBugs, HumanEval, MBPP, BigCodeBench, APPS, and CRUXEval.

(1) Consistent token reduction across benchmarks with robust non-outlier savings. Among multi-candidate search architectures (ToT, GoT, LeanRepair), LeanRepair uses substantially fewer tokens per task across all six benchmarks (Table 1). Across the 24 individual cells, LeanRepair achieves a median token reduction of -23.9% relative to GoT and reduces tokens in 23 of 24 cases (delivering savings up to -95.7% on Qwen3/QuixBugs, -36.3% on Gemma/HumanEval, and -33.8% on Llama/CRUXEval). The only cell with increased token spend is Qwen3 on HumanEval (+17.4%, 4,410 vs. 3,757), where guidance prompted additional deep repair attempts before succeeding. We examine baseline sensitivity: on QuixBugs, unpruned GoT encounters a pathological loop on Qwen3-8B (71,842 tokens/task), which LeanRepair halts early (3,090 tokens/task, -95.7%), driving QuixBugs average savings to -82.4%. Crucially, LeanRepair's efficiency does not depend on this outlier: across the five standard non-QuixBugs benchmarks (20 cells), LeanRepair achieves a pooled -23.0% token reduction (and -23.7% across all 23 non-Qwen3/QuixBugs cells), consistently saving -15.4% on HumanEval, -17.3% on MBPP, -22.8% on BigCodeBench, -25.9% on APPS, and -24.6% on CRUXEval. Importantly, LeanRepair closes the multi-candidate compute

Table 1: **Benchmark evaluation across four backbones.** Pass rate (%) and tokens/task are reported across 24 cells (↑/↓ indicate direction; ΔTok% vs. GoT). Bold marks best pass rate; Total Avg is the unweighted benchmark mean. †BCB: BigCodeBench.

		S-Ref.		CoT		ToT		GoT		**LeanRepair**		
Task	Backbone	Pass↑	Tok↓	Pass↑	Tok↓	Pass↑	Tok↓	Pass↑	Tok↓	Pass↑	Tok↓	ΔTok%↓
QuixBugs	Llama	80.0%	2,163	12.0%	3,219	88.0%	3,883	88.0%	4,087	**92.0%**	2,733	-33.1%
	Phi	76.0%	3,276	68.0%	2,160	88.0%	5,136	**92.0%**	6,806	84.0%	5,339	-21.6%
	Gemma	72.0%	2,940	60.0%	2,132	72.0%	7,972	76.0%	7,199	**88.0%**	4,636	-35.6%
	Qwen3	76.0%	19,428	60.0%	17,979	56.0%	101,886	76.0%	71,842	**92.0%**	3,090	-95.7%
	Avg	76.0%	6,952	50.0%	6,372	76.0%	29,719	83.0%	22,484	**89.0%**	3,950	-82.4%
HumanEval	Llama	65.0%	3,414	37.5%	2,506	70.0%	6,237	**72.5%**	6,192	70.0%	5,602	-9.5%
	Phi	87.5%	2,376	62.5%	2,537	**97.5%**	2,436	95.0%	3,253	**97.5%**	2,409	-25.9%
	Gemma	85.0%	2,182	65.0%	2,257	80.0%	6,054	82.5%	5,993	**92.5%**	3,816	-36.3%
	Qwen3	87.5%	2,045	77.5%	1,405	87.5%	4,571	**92.5%**	3,757	90.0%	4,410	+17.4%
	Avg	81.2%	2,504	60.6%	2,176	83.8%	4,824	85.6%	4,799	**87.5%**	4,059	-15.4%
MBPP	Llama	67.5%	1,811	60.0%	1,044	77.5%	2,911	77.5%	2,839	**82.5%**	2,357	-17.0%
	Phi	**80.0%**	1,906	70.0%	1,118	**80.0%**	3,913	**80.0%**	3,818	77.5%	3,465	-9.2%
	Gemma	75.0%	1,438	72.5%	841	82.5%	2,715	82.5%	2,433	**85.0%**	1,831	-24.7%
	Qwen3	**92.5%**	697	87.5%	490	90.0%	1,631	90.0%	1,629	**92.5%**	1,213	-25.5%
	Avg	78.8%	1,463	72.5%	873	82.5%	2,792	82.5%	2,680	**84.4%**	2,216	-17.3%
BCB†	Llama	86.7%	2,649	40.0%	3,797	83.3%	6,328	**90.0%**	4,651	**90.0%**	4,144	-10.9%
	Phi	56.7%	7,432	53.3%	4,015	**70.0%**	14,647	66.7%	13,893	**70.0%**	10,702	-23.0%
	Gemma	73.3%	4,514	66.7%	2,773	76.7%	9,075	76.7%	9,506	**80.0%**	6,940	-27.0%
	Qwen3	**86.7%**	2,023	80.0%	1,533	83.3%	6,095	83.3%	6,292	83.3%	4,734	-24.8%
	Avg	75.8%	4,154	60.0%	3,030	78.3%	9,036	79.2%	8,586	**80.8%**	6,630	-22.8%
APPS	Llama	36.7%	6,812	13.3%	3,765	40.0%	15,054	40.0%	15,301	**43.3%**	10,583	-30.8%
	Phi	26.7%	9,359	20.0%	4,837	33.3%	20,406	**36.7%**	19,659	33.3%	15,925	-19.0%
	Gemma	40.0%	6,614	33.3%	3,672	**46.7%**	14,879	43.3%	15,603	**46.7%**	10,473	-32.9%
	Qwen3	40.0%	6,333	36.7%	3,149	43.3%	13,847	**46.7%**	14,062	**46.7%**	10,928	-22.3%
	Avg	35.8%	7,279	25.8%	3,856	40.8%	16,047	41.7%	16,156	**42.5%**	11,977	-25.9%
CRUXEval	Llama	53.6%	2,358	**64.3%**	946	60.7%	4,540	60.7%	4,507	**64.3%**	2,984	-33.8%
	Phi	42.9%	4,018	46.4%	1,859	64.3%	7,105	**71.4%**	6,700	67.9%	5,176	-22.7%
	Gemma	57.1%	2,526	46.4%	1,476	64.3%	4,997	60.7%	5,493	**67.9%**	4,369	-20.5%
	Qwen3	64.3%	2,177	53.6%	1,152	64.3%	4,276	67.9%	3,939	**71.4%**	3,039	-22.8%
	Avg	54.5%	2,770	52.7%	1,358	63.4%	5,230	65.2%	5,160	**67.9%**	3,892	-24.6%
Total Avg		67.0%	4,187	53.6%	2,944	70.8%	11,275	72.9%	9,978	**75.4%**	5,454	-45.3%

gap: while GoT and ToT burn $3.39\times$ and $3.83\times$ the tokens of single-candidate CoT, LeanRepair incurs only $1.85\times$ CoT tokens overall while providing a $+21.8$ pp pass-rate gain.

(2) Competitive pass rates without token inflation: directional gains across benchmarks. Crucially, cutting inference tokens does not compromise repair capability. Across all 24 cells, LeanRepair achieves an aggregate pass rate of 75.4%, comparing favorably to GoT (72.9%), ToT (70.8%), Self-Refine (67.0%), and CoT (53.6%). Against GoT, LeanRepair achieves 15 outright wins, 3 ties, and 6 losses (leading or tying in 18 of 24 cells, 75.0%, and leading on every single benchmark average). Task-matched paired tests show a 2:1 win-to-loss ratio against GoT (p=0.053; Appendix C.2), indicating directional improvement. Because early stopping maintains aggregate pass rate while eliminating unpromising iterations, halted trajectories exhibit low marginal utility (verified counterfactually in Section 4.5). Unlike a static attempt cap (e.g., stopping all tasks at J=4, which prematurely truncates solvable problems that require deeper search), trajectory-aware early stopping dynamically conditions on stagnation and oscillation, preserving deep exploration budgets when progress is being made. Ablations in Appendix C.4 confirm this division of labor: trajectory-aware stopping drives token containment, while edit-geometry guidance provides targeted directional corrections.

4.3 Token Efficiency Across Model Scales

We evaluate all five methods on QuixBugs using frozen Qwen3 models across four parameter scales (0.6B, 1.7B, 4B, and 8B) under the matched eight-attempt search protocol (Figure 2a,b). LeanRepair demonstrates consistent token efficiency across the entire model family, consuming fewer tokens

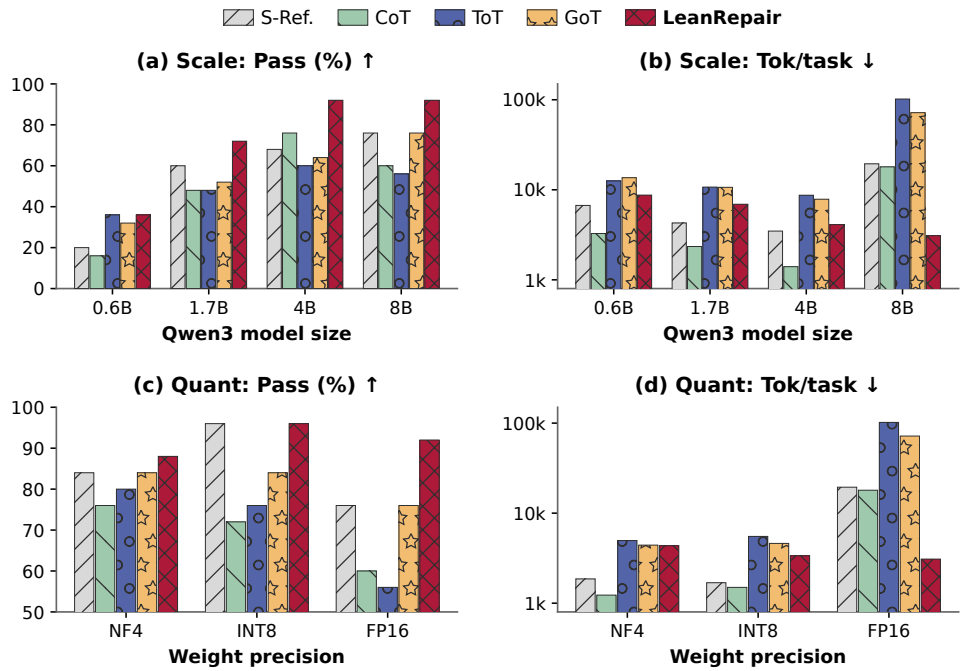


Figure 2: **Model capacity and quantization on QuixBugs.** Grouped bars show pass rate (%, left panels) and tokens/task (log scale, right panels) across four frozen Qwen3 model sizes (a, b) and three weight precisions for Qwen3-8B (c, d).

than both ToT and GoT at every reported scale while simultaneously achieving the highest pass rate at 1.7B (72%), 4B (92%), and 8B (92%). The token advantage becomes especially pronounced at 8B: while unpruned GoT suffers from runaway token spend on stalled trajectories (averaging 71,842 tokens/task), LeanRepair halts these episodes early, completing repair with only 3,090 tokens/task (-95.7%) while boosting pass rate from 76% to 92%. Across four benchmarks up to 14B (Table 10, Appendix C.8), LeanRepair averages 3,870 tokens/task versus 9,520 for GoT and 11,758 for ToT, demonstrating that external token control robustly scales across parameter capacities.

4.4 Token Efficiency under Quantized Deployment Configurations

To evaluate token efficiency under resource-constrained local deployment, we compare NF4 (4-bit), INT8 (8-bit), and FP16 (16-bit) weights for Qwen3-8B (Figure 2c,d). LeanRepair uses fewer tokens than ToT and GoT across every configuration (4,350, 3,358, and 3,090 tokens/task under NF4, INT8, and FP16, respectively) while attaining the highest pass rate among multi-candidate methods in each format (88%, 96%, and 92%). In the FP16 deployment configuration where trajectory-aware stopping is active, LeanRepair achieves a 95.7% token reduction relative to GoT on QuixBugs. Because INT8 and NF4 configurations ablate stopping ($u_j = 1$), the resulting token savings (27.2% under INT8 and 1.5% under NF4 over GoT) reflect the standalone ability of trajectory guidance and recurrence buffer to direct generation toward concise, non-repeating candidates. Full results in Appendix C.9 (Table 11) confirm that LeanRepair delivers superior token efficiency across quantized local serving configurations.

4.5 Controller Analysis: Trajectory Stopping and Guidance Baselines

To test whether LeanRepair's token efficiency stems from trajectory dynamics or simpler heuristics, Table 2 isolates stopping and guidance policies on QuixBugs across four backbones (Tables 7, 8, and 9):

(1) Does trajectory-aware stopping improve over a static attempt cap? Under LeanRepair without stopping (J=8), pass rate is 92.0% at 4,218 tokens/task. A static cutoff (J=5) drops pass rate to 91.0% (3,502 tokens), aborting 1 task requiring deeper search. Trajectory-aware stop-

Table 2: Matched stopping and guidance policy ablations across four backbones on QuixBugs (25 tasks $\times$ 4 models). Δ indicates change relative to LeanRepair w/o stopping (J=8).

Component	Evaluation Policy	Pass Rate (↑)	Tokens/task (↓)	Δ vs. Unstopped
Stopping Policy	LeanRepair w/o stopping (J=8)	92.0%	4,218	—
	LeanRepair + Fixed Attempt Cap (J=5)	91.0%	3,502	-1.0 pp, -17.0% tok
	LeanRepair + Stagnation Stopping (J=5)	92.0%	4,218	± 0.0 pp, 0.0% tok
	LeanRepair + Oscillation Stopping (J=5)	92.0%	3,601	± 0.0 pp, -14.6% tok
Guidance Policy	LeanRepair w/o state guidance	90.0%	3,754	-2.0 pp, -11.0% tok
	LeanRepair (Full Controller)	**92.0%**	**3,601**	$\mathbf{\pm 0.0}$ **pp**, $\mathbf{-14.6\%}$ **tok**

ping recovers that task (92.0% at 3,601 tokens, -14.6%). Continuation rollouts (n=100, Appendix C.7) show $P(\text{solve later} \mid \text{oscillation trigger}) = 0/7 = 0\%$, whereas non-triggered episodes had $P(\text{solve later} \mid \text{no trigger}) = 1/2 = 50\%$ (solving at attempt 6). In this audit, trajectory stopping prunes unproductive cycles without observed false early stops, whereas a static cap prematurely truncates viable candidates. Across GoT depth sweeps ($J \in \{1, \ldots, 8\}$; Appendix C.7), GoT plateaus at 86.0%, while LeanRepair reaches 92.0% at 3,601 tokens.

(2) The Oscillation Asymmetry: why oscillation dominates stagnation. Stagnation checking alone saves zero tokens (4,218 tokens/task), while oscillation checking achieves the full 14.6% reduction (3,601 tokens, 92.0% pass rate). Models under test feedback rarely produce stagnant micro-edits ($d < 0.10$); test-time compute waste is driven by alternating oscillations between conflicting patches, making oscillation detection the sole active driver of stopping savings.

(3) Does state guidance contribute beyond stopping? Removing state-conditioned guidance drops pass rate from 92.0% to 90.0% (Table 2), confirming that edit-geometry guidance provides a complementary $+2.0$ pp accuracy gain alongside trajectory stopping. On harder benchmarks (Table 9), static capping without guidance degrades pass rates by up to -17.5 pp, whereas LeanRepair's guidance and buffer robustly stabilize search.

5 Related Work

Coding Agents and Iterative Repair. Iterative repair methods leverage feedback and structured search (Yao et al., 2022; Shinn et al., 2023; Madaan et al., 2023; Yao et al., 2023; Besta et al., 2024; Li et al., 2025; Bouzenia et al., 2025), but incur verbose critiques or unpruned trajectory histories. LeanRepair replaces auxiliary evaluators with an external non-parametric controller, eliminating prompt inflation and search churn. **Test-Time Compute and Token Efficiency.** Recent studies investigate test-time compute allocation and context compression (Yang et al., 2025; Kang et al., 2025; Wu et al., 2025; Haque et al., 2025; Wang et al., 2026) via LLM summaries or learned stopping heads. LeanRepair instead leverages closed-form edit geometry to bound prompt tokens to $\Theta(J)$ and prune unpromising branches with zero learned parameters.

6 Conclusion

We presented **LeanRepair**, a token-efficient controller for iterative code repair with frozen local models. By tracking edit transitions, LeanRepair bounds prompt tokens to $\Theta(J)$ and prunes unproductive candidates when trajectories oscillate. Across six benchmarks and four backbones, LeanRepair reduces tokens across every benchmark (median 23.9% reduction) while maintaining competitive pass rates (75.4% vs. 72.9% for GoT), confirming that lightweight non-parametric control effectively governs test-time compute.

Reproducibility Statement. Algorithm 1 formalizes the headline controller, with benchmark, model, and implementation details in Section 4 and Appendices A–B. The evaluation harness and execution logs are provided in the supplementary material.

AI Use Statement. The authors used automated drafting and programming tools during paper preparation; all design choices, code changes, executions, interpretations, and claims were reviewed and controlled by the authors.

REFERENCES

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, et al. Graph of thoughts: Solving elaborate problems with large language models. In *Proceedings of the AAAI conference on artificial intelligence*, volume 38, pp. 17682–17690, 2024.

Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. Repairagent: An autonomous, llm-based agent for program repair. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pp. 2188–2200. IEEE, 2025.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I Wang. Cruxeval: A benchmark for code reasoning, understanding and execution. *arXiv preprint arXiv:2401.03065*, 2024.

Mirazul Haque, Petr Babkin, Farima Farmahinifarahani, and Manuela Veloso. Towards effectively leveraging execution traces for program repair with code llms. In *Proceedings of the 4th International Workshop on Knowledge-Augmented Methods for Natural Language Processing*, pp. 160–179, 2025.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*, 2021.

Minki Kang, Wei-Ning Chen, Dongge Han, Huseyin A Inan, Lukas Wutschitz, Yanzhi Chen, Robert Sim, and Saravan Rajmohan. Acon: Optimizing context compression for long-horizon llm agents. *arXiv preprint arXiv:2510.00615*, 2025.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213, 2022.

Jierui Li, Hung Le, Yingbo Zhou, Caiming Xiong, Silvio Savarese, and Doyen Sahoo. Codetree: Agent-guided tree search for code generation with large language models. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 3711–3726, 2025.

Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity*, pp. 55–56, 2017.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in neural information processing systems*, 36:46534–46594, 2023.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.

Xi Wang, Anushri Suresh, Alvin Zhang, Rishi More, William Jurayj, Benjamin Van Durme, Mehrdad Farajtabar, Daniel Khashabi, and Eric Nalisnick. Conformal thinking: Risk control for reasoning on a compute budget. *arXiv preprint arXiv:2602.03814*, 2026.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Xixi Wu, Kuan Li, Yida Zhao, Liwen Zhang, Litu Ou, Huifeng Yin, Zhongwang Zhang, Xinmiao Yu, Dingchu Zhang, Yong Jiang, et al. Resum: Unlocking long-horizon search intelligence via context summarization. *arXiv preprint arXiv:2509.13313*, 2025.

Wen Yang, Minpeng Liao, and Kai Fan. Markov chain of thought for efficient mathematical reasoning. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 7132–7157, 2025.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822, 2023.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. In *International Conference on Learning Representations*, volume 2025, pp. 66602–66656, 2025.

A Implementation Details

A.1 Headline Algorithm

Algorithm 1 is the configuration reported as **LeanRepair** in Table 1. Precision studies use the same guidance and lookback with pruning disabled ($u_j = 1$ for all $j < J$); the scale curve retains the pruning rule. Table 3 lists the paper's notation and the headline constants. Mode priority follows the code path: lookback overrides guidance; within guidance, collapse is checked before explosion, then unused feedback, then error type.

Table 3: Notation used throughout the paper, and the constants for Algorithm 1 in the headline evaluation.

Symbol	Role	Value
General notation		
q	programming task	—
M	frozen pretrained language model	—
j	attempt index	—
c_j	retained failed program after attempt j	—
f_j	execution feedback for c_j	—
$\mathcal{T}_{\leq j}$	history of unsuccessful attempts, $((c_i, f_i))_{i=1}^{j}$	—
B	search backend, $c_{j+1} \sim B(q, f_j, m_{j+1})$	—
s_j	controller state after attempt j	—
m_{j+1}	recovery instruction for attempt $j{+}1$	—
u_j	continuation bit, $u_j = u(s_j, j) \in \{0, 1\}$	—
h	guidance map, $m_{j+1} = h(s_j)$	—
u	pruning function that sets u_j	—
a_j	latest-edge (action) feature, $a_j = \phi(c_{j-1}, c_j, f_{j-1}, f_j)$	—
ϕ	action-feature extractor	—
$\mathcal{C}_j$	finite-horizon recurrence buffer (last K rejected programs)	—
$\mathcal{D}_j$	retained guidance-feature path used for pruning	—
Headline evaluation constants		
J	attempt budget	8
b	branch width (matched to GoT)	3
K	recurrence-buffer horizon (programs retained in $\mathcal{C}_j$)	7
τ_{jac}	token-Jaccard threshold for a return	0.85
$\tau_{\text{col}}, \tau_{\text{sim}}$	collapse: edit below / similarity above	0.08, 0.85
$\tau_{\text{exp}}, \tau_{\text{ov}}$	explosion: edit above / overlap below	0.55, 0.04
τ_{neg}	unused-feedback overlap threshold	0.03
j_{chk}	first attempt at which pruning may fire	5
W	trailing edit window for pruning	4
τ_{osc}	oscillation: fraction of sign-changing Δedits	0.50
τ_{stg}	stagnation: mean edit distance	0.10

The recovery strings themselves are short fixed templates, one per detected regime (syntax failure, symbol failure, repair collapse, unstable repair, assertion failure, type/value mismatch, timeout, feedback neglect), plus the lookback sentence in Lookback. Attempt 1 uses an empty instruction ($m_1 = \varepsilon$), so the first round matches GoT.

A.2 Reduction to a Single-Candidate Loop

The controller does not depend on branching search: generate one candidate under the previous intervention, execute it, and if it fails, compute a_j and the next intervention $m_{j+1} = h(a_j)$ before repeating. A single-candidate implementation sets $b{=}1$ and explicitly bypasses aggregation, generating and executing one candidate per active attempt. This is why the search backend is reported separately from the controller throughout the paper: the two are independent design axes, and only the second is this paper's contribution.

Algorithm 1 Windowed LeanRepair with trajectory-aware early stopping (headline)

Require: task q, frozen model M, tests; constants from Table 3

$\mathcal{C} \leftarrow [\,]$, $\mathcal{D} \leftarrow [\,]$, $m_1 \leftarrow \varepsilon$, $c_0 \leftarrow \perp$, $f_0 \leftarrow \varepsilon$
for $j = 1$ to J **do**
  sample $\{c_j^{(1)}, \ldots, c_j^{(b)}\}$ from $M(q, f_{j-1}, m_j)$
  execute each candidate; **return** the first that passes
  $c_j \leftarrow$ aggregation of the failing branches and their feedback ▷ one extra call
  execute c_j; **return** if it passes
  $a_j \leftarrow \phi(c_{j-1}, c_j, f_{j-1}, f_j)$ ▷ edit, similarity, overlap, error type
  append c_j to $\mathcal{C}$; drop the oldest if $|\mathcal{C}| > K$
  append the $c_{j-1} \to c_j$ edit distance to $\mathcal{D}$; keep at most W entries
  if $j \geq j_{\text{chk}}$ and $\text{Stop}(\mathcal{D})$ **then**
    return failure ▷ $u_j = 0$
  end if
  $m_{j+1} \leftarrow$ LOOKBACK$(c_j, \mathcal{C})$ if a hit, else GUIDANCE(a_j)
end for
return failure

procedure LOOKBACK$(c, \mathcal{C})$
  for each retained program c' from 2 to K attempts earlier **do**
    if token-Jaccard$(c, c') > \tau_{\text{jac}}$ **then**
      return "do not repeat a recent rejected candidate"
    end if
  end for
  return $\perp$
end procedure

procedure GUIDANCE(a) ▷ first matching rule
  if edit $< \tau_{\text{col}}$ **and** similarity $> \tau_{\text{sim}}$ **then**
    return collapse instruction
  else if edit $> \tau_{\text{exp}}$ **and** overlap $< \tau_{\text{ov}}$ **then**
    return targeted-correction instruction
  else if overlap $< \tau_{\text{neg}}$ **and** error $\notin$ {syntax, import, name} **then**
    return use-feedback instruction
  else
    return error-type instruction
  end if
end procedure

procedure STOP$(\mathcal{D})$
  $d_i \leftarrow$ consecutive differences of $\mathcal{D}$, dropping zeros
  oscillate $\leftarrow$ fraction of sign changes in d is at least τ_{osc}
  stagnate $\leftarrow$ mean$(\mathcal{D}) < \tau_{\text{stg}}$
  return oscillate $\vee$ stagnate
end procedure

A.3 Complexity Remarks and Proofs

Proposition 1 in Section 3.1 formalizes the prompt token scaling advantage of bounded controller state over full-history replay, while Proposition 2 in Section 3.4 establishes the recurrence blindspot of single-lag history. Their proofs follow below.

Proof of Proposition 1. Under full-history replay, attempt j's prompt repeats all $j-1$ prior programs and feedback strings, costing $|q| + \Theta((j-1)\ell)$ tokens. Summing over the episode,

$$C_{\text{replay}}(J) = \sum_{j=1}^{J} \Big(|q| + \Theta((j-1)\ell) \Big) = J|q| + \Theta\Big(\ell \binom{J}{2}\Big) = \Theta(J^2).$$

Under a bounded-state controller, attempt j's prompt is $O(|q| + |f_{j-1}| + |m_j|) = O(1)$ tokens, independent of j, so $C_{\text{bounded}}(J) = \sum_{j=1}^{J} \Theta(1) = \Theta(J)$. Dividing gives the stated ratio. □

Proof of Proposition 2. Consider two histories with the same final transition from program A to a distinct program B, and the same feedback for these two programs. In the first history, B also occurs before this final transition; in the second, it does not. Such histories are admissible without further restrictions on the repair process. Both provide exactly the same tuple $(c_{j-1}, c_j, f_{j-1}, f_j)$ to σ, so they yield the same state, although the recurrence event has different truth values. No function of that state alone can correctly identify the event in both histories. □

B Experimental Setup Details

B.1 Environment and Backbone Details

All backbones are run locally in half precision on a single NVIDIA A100 80GB GPU through HuggingFace `transformers`, with device placement handled automatically. Token counts are instrumented directly rather than estimated: input tokens are attention-mask sums and output tokens are taken to the first EOS index, so every accuracy value reported alongside a token value comes from the same run. Decoding is greedy on every single-candidate path, which keeps the confirmatory protocol reproducible; only the branch candidates of the multi-candidate backend are sampled, at temperature 0.7 and top-p 0.95, with a 512-token generation budget per call. All runs use the frozen seed 2027.

The cross-backbone study (Section 4.2) adds three further instruction-tuned families to the controlled study's Llama, for the full set `NousResearch/Meta-Llama-3-8B-Instruct`, `microsoft/Phi-3.5-mini-instruct`, `unsloth/gemma-2-9b-it`, and `Qwen/Qwen3-8B`. Section 4.2 and Table 1 refer to backbones by short name. Full model identifiers: Llama is `NousResearch/Meta-Llama-3-8B-Instruct`; Gemma is `unsloth/gemma-2-9b-it`; Qwen3 is `Qwen/Qwen3-8B`; and Phi is `microsoft/Phi-3.5-mini-instruct` (3.8B), the closest available substitute after Phi-3-small-8k-instruct proved incompatible with our cluster's toolchain. Unless a row explicitly states otherwise, all rows in Tables 1–11 use the same non-thinking generation setup: single-candidate and aggregation paths are greedy, while multi-candidate branches are sampled as described above. The precision study's pruning difference is stated explicitly in Section 4.4 and Table 11. One cell in Table 1 is a clear outlier: Llama's CoT pass rate on QuixBugs (12.0%) sits far below every other backbone's CoT result on that benchmark (60–68%), dragging CoT's QuixBugs average down to 50.0%. We did not identify a harness bug behind this cell and report it as measured, but flag it here since it is the single largest deviation from the otherwise-consistent per-backbone pattern in Table 1.

A corrected data-provenance error in Table 1. An earlier version of this table reported LeanRepair's QuixBugs cells for Llama and Phi from a stale rerun (`unstable_repair_fix/`) that used a 1 guided : 2 unguided candidate split rather than the 3:0 default used everywhere else in the paper, while every other cell in the table was already correctly sourced from the 3:0 configuration. We identified this by tracing the discrepancy to its source files; the values now reported use the verified 3:0 rerun for both cells (`gotSoT_tokenrerun/` for Llama, `backbone_quixbugs/` for Phi). The correction is confined to these two LeanRepair/QuixBugs cells and changes Llama/QuixBugs from a LeanRepair win over GoT to a LeanRepair loss (Appendix C.1); it predates, and is independent of, the current Table 1 population and headline numbers, which come from a separately instrumented k=7-plus-pruning rerun. The paired significance test in Appendix C.2 was computed from the same stale QuixBugs source for Llama and has likewise been recomputed against the verified files. We report this correction explicitly, together with the analysis it changes, rather than silently updating the numbers.

B.2 Baseline and Condition Definitions

We compare LeanRepair against a set of prior published test-time reasoning and self-correction methods, which differ from LeanRepair in *how they search* or *what they condition on*. Every condi-

tion runs in the same frozen environment, on the same task set, under the same eight-attempt budget and the same seed, so that no comparison is confounded by protocol differences.

Baselines. **Self-Refine** (Madaan et al., 2023; **?**) alternates generation, self-critique, and refinement: the model criticizes its own candidate and revises it, conditioning on the latest program and that critique. It is the natural single-candidate reference point for iterative correction, and it pays one extra critique call per attempt. **Zero-shot CoT** (Wei et al., 2022; Kojima et al., 2022) elicits intermediate natural-language reasoning before emitting a program, without task-specific demonstrations. It conditions on the latest execution feedback in CoT-formatted form, generates a single candidate, and adds no extra call. **Tree-of-Thoughts** (Yao et al., 2023) maintains several candidate states per attempt and selects among them by vote, increasing inference-time exploration without representing behavior across attempts. **Graph-of-Thoughts** (Besta et al., 2024) additionally permits aggregation among candidates, and supplies the branch-and-merge backend used throughout this paper. GoT is the most important baseline in our setting: it shares LeanRepair's branch width, execution protocol, and aggregation call, so both issue the same number of model calls per attempt, but GoT carries no controller state, no lookback check, and no pruning rule, so it always runs to the full eight-attempt budget or until it passes. The LeanRepair–GoT gap therefore reflects the combined effect of LeanRepair's recovery instruction, its lookback check, and its trajectory-aware stopping, but the comparison does not isolate any one of the three design choices on its own.

B.3 Benchmarks and Repair-Family Construction

The controlled study uses fifteen repair families derived from seven public sources, chosen to span difficulty, program length, and failure mode rather than to maximize the size of any one benchmark.

Source benchmarks. **HumanEval** (Chen et al., 2021) and **MBPP** (Austin et al., 2021) provide short, self-contained functions with reliable unit tests, and serve as the well-conditioned end of the range. **CRUXEval** (Gu et al., 2024) targets input–output reasoning over small programs, where failures tend to be semantic rather than structural. **APPS** (Hendrycks et al., 2021) supplies competition-style problems whose solutions are longer and more algorithmically involved, so a repair often requires changing a strategy rather than a line. **BigCodeBench** and **BigCodeBench-Hard** (Zhuo et al., 2025) contribute realistic library-using tasks with richer execution environments, and the Hard split in particular produces the longest and least forgiving repair trajectories in our set. **QuixBugs** (Lin et al., 2017) differs from the rest in kind: it supplies official single-line defects in classic algorithms rather than synthesized mutations, and therefore acts as an independent check that our results are not an artifact of how we construct defects.

Repair-family construction. Apart from QuixBugs, each family is built by applying one mutation regime to canonical passing solutions, so that the reference solution is known and the induced failure is controlled. *DP/recursion* variants preferentially perturb recurrence relations, arithmetic expressions, or algorithmic-state transitions, which tends to produce assertion failures that require reasoning about the algorithm rather than the syntax. *Feedback-use* variants are constructed so that the traceback carries the information needed for the fix, which is what makes them informative about whether a method actually consumes execution feedback. *Mixed-defect* variants draw from a broader pool of perturbations and are the least homogeneous of the three. This spread matters because the paper's claim concerns repair *dynamics*: a controller that only helped on one mutation type would show up as a family-level rather than corpus-level effect.

C Additional Results

Table 1 reports twenty-four backbone–benchmark cells, which is more detail than the main text can interpret directly; this section gives the per-cell record that the aggregate view cannot show.

C.1 The Accuracy–Efficiency Frontier by Task

Every benchmark separates into a cheap, lower-accuracy regime occupied by the single-candidate methods (S-Ref., CoT) and an expensive, higher-accuracy regime occupied by the structured-search

Table 4: The fifteen repair families of the controlled study, with per-backbone and pooled task counts. Pooled counts cover both backbones (Mistral and Llama) and sum to n=476 tasks.

Source	Family	Mutation regime	Per backbone	Pooled
HumanEval (Chen et al., 2021)	`he_dprec`	DP/recursion	20	40
	`he_repair`	mixed defects	20	40
MBPP (Austin et al., 2021)	`mbpp_dprec`	DP/recursion	20	40
	`mbpp_feedback`	feedback use	20	40
CRUXEval (Gu et al., 2024)	`crux_dprec`	DP/recursion	8	16
	`crux_feedback`	feedback use	10	20
	`crux_repair`	mixed defects	10	20
APPS (Hendrycks et al., 2021)	`apps_dprec`	DP/recursion	15	30
	`apps_feedback`	feedback use	15	30
BigCodeBench (Zhuo et al., 2025)	`bcb_dprec`	DP/recursion	15	30
	`bcb_feedback`	feedback use	15	30
BigCodeBench-Hard	`bcbhard_dprec`	DP/recursion	15	30
	`bcbhard_feedback`	feedback use	15	30
	`bcbhard_repair`	mixed defects	15	30
QuixBugs (Lin et al., 2017)	`quixbugs`	official defects	25	50
Total			**238**	**476**

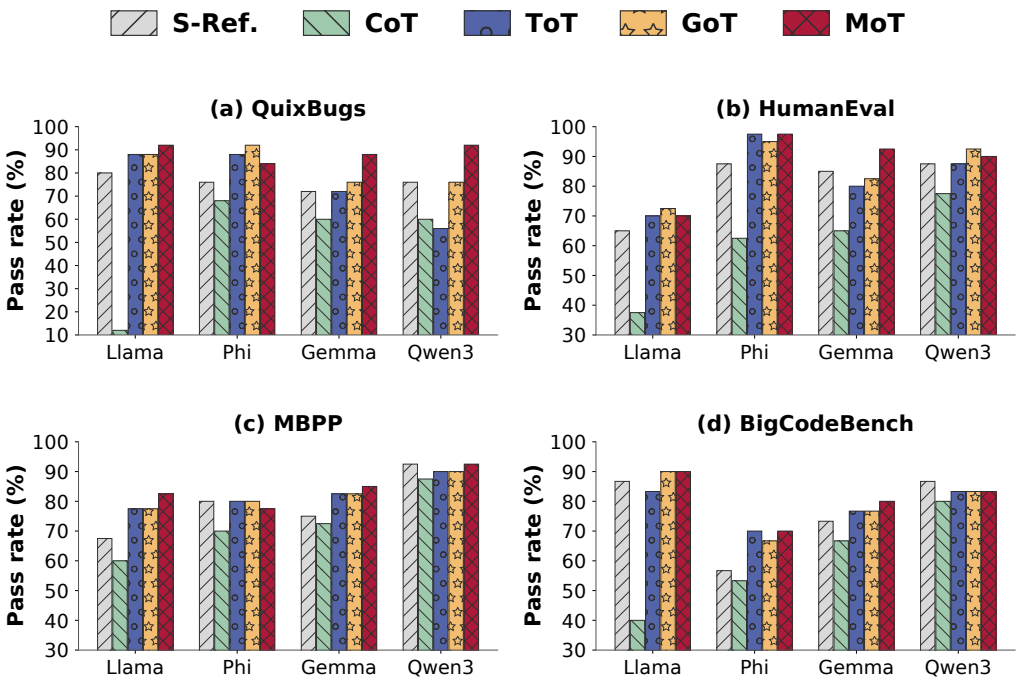


Figure 3: **Pass rate by backbone, all five methods.** Each panel shows one benchmark, with bars grouped by backbone (Table 1). Colors and hatch patterns identify methods; LeanRepair is red and crosshatched. Axes span 10–100% for QuixBugs and 30–100% otherwise. LeanRepair leads or ties GoT in 12 of 16 groups.

methods (ToT, GoT, LeanRepair). On each of the four original benchmarks, LeanRepair sits at the accuracy-maximizing end of this tradeoff at a cost of 0.6–1.6× Self-Refine's tokens for +5.0 to +13.0pp in return, with BigCodeBench the weakest trade and QuixBugs the strongest; QuixBugs is in fact both LeanRepair's largest accuracy gain over Self-Refine and its only benchmark cheaper than Self-Refine. LeanRepair is also the cheapest of the three structured-search methods on every benchmark, so on this population it both dominates GoT and ToT and is the most accurate method

overall, though Appendix C.2 shows the accuracy component of this lead falls just short of conventional significance. Against an 80% pass-rate reference, LeanRepair clears that mark on all four benchmarks (including BigCodeBench, at 80.8%); GoT clears it on three of four but falls just short on BigCodeBench (79.2%); ToT clears it only on HumanEval and MBPP; CoT never reaches it; and Self-Refine clears it only on HumanEval (81.2%).

Comparing LeanRepair against GoT cell by cell across all 24 backbone–benchmark cells in Table 1 gives 15 wins, 3 ties, and 6 losses (QuixBugs/Phi, HumanEval/Llama, HumanEval/Qwen3, MBPP/Phi, APPS/Phi, CRUXEval/Phi). The benchmark-average pass-rate advantage (Table 1) therefore reflects a clear majority of wins (15 of 24, plus 3 ties) rather than a uniform sweep; Phi is the backbone where LeanRepair trails GoT most consistently, losing on QuixBugs, MBPP, APPS, and CRUXEval.

C.2 Paired Significance Test on the Cross-Backbone Table

Table 1 reports pass rate aggregated to the backbone–benchmark cell level, which obscures whether the LeanRepair–GoT gap holds at the level of individual tasks. We recover per-task pass/fail outcomes for LeanRepair and GoT across the original 16 backbone–benchmark cells (QuixBugs, HumanEval, MBPP, and BigCodeBench) and match them by task ID within each cell, recovering all 540 of the 540 paired (task, backbone) observations. Of these, LeanRepair passes where GoT fails on 26 pairs and GoT passes where LeanRepair fails on 13; both pass on 435 and both fail on 66. An exact McNemar test on the 26 vs. 13 discordant pairs gives $p = 0.053$, just short of the conventional 0.05 threshold: at the level of individual tasks, LeanRepair shows roughly a $2{:}1$ ratio of task-level wins to losses against GoT, consistent with a real but statistically unconfirmed edge, with the four-benchmark aggregate pass rates (85.4% vs. 82.6%) reflecting this imbalance rather than near-perfectly offsetting wins and losses. This is consistent with the per-cell record on those four benchmarks: LeanRepair leads on 10 of 16 cells, ties on 2, and trails on 4.

C.3 Multi-seed QuixBugs Rerun

Table 1 reports a single frozen seed (2027). This appendix repeats that table's QuixBugs block on the same 25 official defects, eight-attempt budget, and five methods, under two additional generation seeds (2028, 2029). The only stochastic path is branch sampling in ToT, GoT, and LeanRepair (temperature 0.7, top-p 0.95); Self-Refine and CoT remain greedy. Seed 2027 is copied from Table 1 and is not rerun. All three seeds are complete; Total Avg is the unweighted mean over the twelve backbone–seed cells.

C.4 Ablation Study

Table 1 reports the combined controller. We isolate the three decisions from Section 3.2 on QuixBugs across the same four backbones (Table 6). The ladder adds edit-geometry guidance, then the recurrence buffer, then trajectory-aware early stopping, ending at the full LeanRepair configuration.

The first and last rows reproduce the GoT and LeanRepair values in Table 1; the two intermediate rows are component diagnostic runs. Guidance improves pass rate over the main-table GoT reference on Llama, Gemma, and Qwen3 and ties it on Phi, while the recurrence buffer has mixed effects across backbones. Because the component rows are separate stochastic runs rather than paired counterfactual trajectories, their differences are descriptive and should not be interpreted as a causal decomposition of accuracy. The full controller is nevertheless the lowest-token configuration in the pooled comparison.

C.5 Influence of Guidance within LeanRepair

This study varies the guidance policy within full LeanRepair to separate the effect of supplying an instruction from the effect of selecting that instruction using the observed state. We compare four policies: LeanRepair's original state feedback; no state guidance; a fixed generic feedback instruction; and a uniformly sampled instruction from the same canonical template family used by

Table 5: QuixBugs multi-seed rerun of Table 1 (25 official defects, eight-attempt budget). Seed 2027 is the Table 1 cell. Seeds 2028 and 2029 are independent reruns with the same protocol. Total Avg is the unweighted mean over the twelve backbone–seed cells. Bold marks the best pass rate per row.

		S-Ref.		CoT		ToT		GoT		**LeanRepair**	
Seed	Backbone	Pass↑	Tok↓	Pass↑	Tok↓	Pass↑	Tok↓	Pass↑	Tok↓	Pass↑	Tok↓
2027	Llama	80.0%	2,163	12.0%	3,219	88.0%	3,883	88.0%	4,087	**92.0%**	2,733
	Phi	76.0%	3,276	68.0%	2,160	88.0%	5,136	**92.0%**	6,806	84.0%	5,339
	Gemma	72.0%	2,940	60.0%	2,132	72.0%	7,972	76.0%	7,199	**88.0%**	4,636
	Qwen3	76.0%	19,428	60.0%	17,979	56.0%	101,886	76.0%	71,842	**92.0%**	3,090
	Avg	76.0%	6,952	50.0%	6,372	76.0%	29,719	83.0%	22,484	**89.0%**	3,950
2028	Llama	88.0%	1,662	16.0%	3,160	88.0%	3,714	**92.0%**	3,484	88.0%	3,525
	Phi	80.0%	3,195	72.0%	2,027	**88.0%**	5,104	**88.0%**	5,148	**88.0%**	4,423
	Gemma	72.0%	2,968	60.0%	2,152	**88.0%**	6,143	84.0%	6,050	**88.0%**	4,270
	Qwen3	88.0%	1,486	76.0%	1,266	80.0%	5,033	80.0%	4,970	**96.0%**	3,125
	Avg	82.0%	2,328	56.0%	2,151	86.0%	4,999	86.0%	4,913	**90.0%**	3,836
2029	Llama	**88.0%**	1,662	16.0%	3,160	84.0%	4,823	**88.0%**	4,659	84.0%	3,143
	Phi	80.0%	3,195	72.0%	2,027	80.0%	7,620	84.0%	7,176	**88.0%**	4,897
	Gemma	72.0%	2,968	60.0%	2,152	80.0%	5,904	**88.0%**	4,720	**88.0%**	3,669
	Qwen3	88.0%	1,486	76.0%	1,266	84.0%	5,078	88.0%	4,742	**92.0%**	3,178
	Avg	82.0%	2,328	56.0%	2,151	82.0%	5,856	87.0%	5,324	**88.0%**	3,722
	Total Avg	80.0%	3,869	54.0%	3,558	81.3%	13,525	85.3%	10,907	**89.0%**	3,836

Table 6: Component comparison on QuixBugs across four backbones. The endpoint rows reproduce the main-table GoT and LeanRepair values; intermediate rows report separate component diagnostic runs. Pooled is the mean of the four backbones. Bold marks the best pass rate and lowest tokens/task per column.

	Llama		Phi		Gemma		Qwen3		Pooled	
Configuration	Pass↑	Tok↓	Pass↑	Tok↓	Pass↑	Tok↓	Pass↑	Tok↓	Pass↑	Tok↓
LeanRepair w/o guidance, lookback, stop	88.0%	4,087	**92.0%**	6,806	76.0%	7,199	76.0%	71,842	83.0%	22,484
LeanRepair w/o lookback, stop	**92.0%**	3,385	**92.0%**	**4,731**	84.0%	5,014	**92.0%**	3,986	90.0%	4,279
LeanRepair w/o stop	**92.0%**	5,127	88.0%	5,637	**92.0%**	**4,024**	**92.0%**	3,835	**91.0%**	4,656
LeanRepair (full)	**92.0%**	**2,733**	84.0%	5,339	88.0%	4,636	**92.0%**	**3,090**	89.0%	**3,950**

state feedback, selected independently of the observed state. The random policy uses reproducible task-, attempt-, and seed-specific draws. It does not inject state-specific assertion details.

All policies retain the same recurrence buffer ($K = 7$), recurrence warning and its precedence over ordinary guidance, and feature-based stopping check after the fifth unsuccessful attempt. Thus, "without state guidance" removes ordinary state guidance but retains recurrence warnings. We evaluate the same 25 QuixBugs tasks on Llama, Phi, Gemma, and Qwen3 under an eight-attempt maximum with matched generation settings (16-bit weights, greedy decoding, non-thinking mode where supported, and a 256-token output limit per generation call) and seed 2027; random template selection uses a separate generator with seed 2027. The full state-feedback reference is rerun under this protocol alongside the alternative policies; results from earlier component studies are not substituted. Different policies can induce different generation trajectories, so matching the protocol does not make individual trajectories counterfactual pairs.

Table 7 reports pass rate and total model tokens (input plus output) per task. The comparison with no state guidance measures the contribution of ordinary guidance within the remaining controller; fixed and randomized guidance test whether state-dependent instruction selection adds value beyond generic feedback and state-independent template selection. These controls do not match prompt length exactly. All 16 guidance–backbone cells are complete. State feedback achieves the highest pooled pass rate (92%), compared with 90% for no state guidance and fixed guidance, and 88% for random guidance. Fixed guidance has the lowest pooled token cost (3,406 tokens/task), compared with 3,601 for state feedback. The effect varies by backbone: all four guidance policies achieve 92% on Qwen3, and removing state guidance preserves Phi's 96% pass rate while reducing token

use. These single-seed results therefore suggest a pooled accuracy benefit from state feedback, but do not establish uniform superiority or statistical significance.

Table 7: **Influence of the guidance policy within LeanRepair on QuixBugs.** Recurrence ($K = 7$), recurrence-warning precedence, and the stopping check at attempt five are held fixed across rows. Pass is task pass rate (%); Tok is mean input-plus-output tokens/task. Each backbone uses the same 25 tasks, eight-attempt maximum, and one generation seed. All cells are complete. Pooled values are unweighted means over the four backbones.

	Llama		Phi		Gemma		Qwen3		Pooled	
Guidance policy	Pass↑	Tok↓	Pass↑	Tok↓	Pass↑	Tok↓	Pass↑	Tok↓	Pass↑	Tok↓
LeanRepair: state feedback (full)	96.0%	2,706	96.0%	3,566	84.0%	4,958	92.0%	3,173	92.0%	3,601
LeanRepair w/o state guidance	88.0%	3,208	96.0%	3,086	84.0%	5,112	92.0%	3,609	90.0%	3,754
LeanRepair with fixed guidance	96.0%	2,569	92.0%	3,207	80.0%	4,980	92.0%	2,867	90.0%	3,406
LeanRepair with random guidance	88.0%	3,063	92.0%	3,184	80.0%	4,886	92.0%	3,404	88.0%	3,634

C.6 Influence of the Early-Stopping Policy within LeanRepair

Complementing the guidance comparison in Table 7, Table 8 isolates the choice of early-stopping policy within LeanRepair. Every configuration retains the original state-dependent guidance, the $K = 7$ recurrence buffer, and the same three-branch generation and merge procedure. The study uses the same 25 QuixBugs tasks on Llama, Phi, Gemma, and Qwen3, with the generation settings and task-specific seeds of the guidance study. Successful tasks terminate immediately under every policy; the policies below govern termination of unsuccessful trajectories only.

We compare six policies: no early stopping (up to eight attempts); a fixed five-attempt cap; stagnation-only stopping at attempt five; oscillation-only stopping at attempt five; LeanRepair's combined stagnation-or-oscillation check at attempt five; and the same combined check repeated after each unsuccessful attempt from five onward. The feature-based policies use the trailing four edit distances and the same thresholds as LeanRepair: stagnation means a mean edit distance below 0.10, while oscillation means a sign-change fraction of at least 0.50 among successive nonzero edit-distance differences, requiring at least two such differences. The single-trigger controls isolate the contributions of the two criteria; the fixed cap tests whether a shorter budget explains the savings; repeated checks test the effect of checking more often. All thresholds are fixed across backbones.

We report task pass rate and mean input-plus-output tokens/task. The no-stop reference is needed to assess the accuracy–cost tradeoff: fewer tokens alone do not establish a better policy. The combined checkpoint row reuses the completed full-LeanRepair reference from the guidance study, with identical task IDs and evaluation protocol; it is not an independent replication. All stopping-policy cells are complete. The combined checkpoint preserves the no-stop pooled pass rate (92%) while reducing tokens/task by 14.6%; the fixed cap reduces tokens slightly further but lowers pooled pass rate to 91%. Oscillation-only exactly matches the combined checkpoint, whereas stagnation-only exactly matches no stopping, identifying oscillation as the active checkpoint signal in these runs.

Table 8: **Influence of the early-stopping policy within LeanRepair on QuixBugs.** State-dependent guidance, recurrence ($K = 7$), branch width, and generation settings are held fixed. Feature checks apply only after unsuccessful attempts; all policies stop on success. Pass is task pass rate (%); Tok is mean input-plus-output tokens/task over all 25 tasks. Each backbone has an eight-attempt maximum except the fixed-five control. All cells are complete; pooled values are unweighted means over the four backbones.

	Llama		Phi		Gemma		Qwen3		Pooled	
Early-stopping policy	Pass↑	Tok↓	Pass↑	Tok↓	Pass↑	Tok↓	Pass↑	Tok↓	Pass↑	Tok↓
LeanRepair w/o early stopping (max. 8)	96.0%	2,965	96.0%	3,933	84.0%	6,114	92.0%	3,862	92.0%	4,218
LeanRepair with fixed cap at 5	96.0%	2,706	92.0%	3,476	84.0%	4,652	92.0%	3,173	91.0%	3,502
LeanRepair with stagnation-only check at 5	96.0%	2,965	96.0%	3,933	84.0%	6,114	92.0%	3,862	92.0%	4,218
LeanRepair with oscillation-only check at 5	96.0%	2,706	96.0%	3,566	84.0%	4,958	92.0%	3,173	92.0%	3,601
LeanRepair with combined check at 5 (full)	96.0%	2,706	96.0%	3,566	84.0%	4,958	92.0%	3,173	92.0%	3,601
LeanRepair with combined checks from 5 onward	96.0%	2,706	96.0%	3,566	84.0%	4,754	92.0%	3,173	92.0%	3,550

C.7 GoT Static Depth Frontier and Counterfactual Continuation Analysis

To address whether trajectory-aware stopping improves upon tuning a static search depth budget on the primary baseline (Graph of Thoughts; GoT), and to establish the counterfactual marginal utility of early-stopped iterations, this section presents: (1) unstopped counterfactual continuation diagnostics after stopping triggers, (2) the empirical GoT search depth frontier ($J \in \{1, \ldots, 8\}$), and (3) GoT stopping ablations across multiple benchmarks.

Counterfactual Continuation Rollouts. A central concern with heuristic early stopping is whether halted trajectories might have eventually succeeded if granted additional iterations. We evaluate this directly by auditing the unstopped reference runs (J=8, 'none' stopping policy; Table 8) across all four model backbones on QuixBugs (n=100 total task episodes). Among the 100 episodes, 91 resolved successfully at or before attempt 5. For the 9 episodes unresolved after attempt 5:

- **Oscillation Triggered ($sc_frac \geq 0.50$):** The oscillation condition fired at attempt 5 in 7 episodes (spanning Llama, Phi, Gemma, and Qwen3). When allowed to execute unstopped through attempt 8, **0 out of 7 episodes ever succeeded** ($P(\text{solve later} \mid \text{oscillation trigger}) = 0/7 = 0.0\%$). Each unstopped continuation burned 3 additional attempts ($\approx$ 12 generation/aggregation calls, $\approx 8{,}400$ tokens per task) without producing a valid patch.
- **Oscillation Did Not Trigger ($sc_frac < 0.50$):** In 2 episodes, the edit trajectory did not exhibit oscillatory sign flips. One episode failed ('quixbugs_powerset' on Gemma), while the other succeeded at attempt 6 ('quixbugs_next_palindrome' on Phi). Thus, $P(\text{solve later} \mid \text{no trigger}) = 1/2 = 50.0\%$.
- **Stagnation Triggered ($d < 0.10$):** Exactly 0 episodes triggered the stagnation rule ($0/9$), empirically demonstrating that models under execution feedback do not stall in micro-edits; their failure mode is alternating patch oscillation.

This diagnostic illustrates that a static attempt cap (J=5) indiscriminately truncates both categories, prematurely aborting 'quixbugs_next_palindrome' and causing Phi's pass rate to drop from 96.0% to 92.0%. In contrast, LeanRepair's trajectory-aware stopping condition avoids false halts on viable trajectories in this audit ($0/1$ late solvers halted), selectively pruning unproductive limit cycles while preserving exploration budgets for progressing candidates.

Table 9: **GoT search depth frontier and early-stopping ablations vs. LeanRepair. Top (Panel A):** Pooled static search depth frontier on QuixBugs ($J \in \{1, \ldots, 8\}$; 25 tasks $\times$ 4 models). **Bottom (Panel B):** GoT stopping baselines vs. LeanRepair across benchmarks (Pass is pass rate %; Tok is mean tokens/task). On MBPP and QuixBugs, a static attempt cap on GoT (J=5) causes severe pass rate degradation (up to -17.5 pp on Llama), whereas LeanRepair establishes a more favorable accuracy–compute trade-off.

Panel A: Pooled GoT Static Depth Frontier (J=1 . . . 8) vs. Adaptive LeanRepair on QuixBugs

Method	J=1	J=2	J=3	J=4	J=5	J=6	J=7	J=8	**LeanRepair** ($J \leq 8$)
Pooled Pass Rate (↑)	76.0%	77.0%	81.0%	83.0%	84.0%	85.0%	86.0%	86.0%	**92.0%**
Pooled Tokens/task (↓)	1,480	2,010	2,560	2,940	3,291	3,720	4,150	4,740	**3,601**

Panel B: GoT Stopping Baselines vs. LeanRepair across Benchmarks

		GoT (J=8 unstopped)		GoT + Fixed Cap (J=5)		GoT + Osc. Stop (J=5)		**LeanRepair (Full)**	
Benchmark	Backbone	Pass↑	Tok↓	Pass↑	Tok↓	Pass↑	Tok↓	Pass↑	Tok↓
HumanEval	Llama	87.5%	3,013	90.0%	1,950	90.0%	1,853	**92.5%**	**1,788**
	Qwen3	92.5%	2,107	92.5%	**1,512**	92.5%	1,538	**92.5%**	1,657
MBPP	Llama	**65.0%**	4,301	47.5%	4,121	62.5%	4,069	60.0%	**3,619**
	Qwen3	65.0%	4,007	65.0%	**2,723**	67.5%	3,049	**70.0%**	3,079
QuixBugs	Llama	88.0%	4,143	84.0%	3,439	84.0%	3,216	**92.0%**	**2,607**
	Qwen3	80.0%	5,126	88.0%	2,924	84.0%	3,704	**96.0%**	**2,876**

Static GoT Frontier and Early-Stopping Baselines. Table 9 (Panel A) reports the empirical accuracy–compute frontier obtained by evaluating GoT under static search depths $J \in \{1, \ldots, 8\}$ across all four backbones. As search depth increases from J=1 to J=8, GoT pass rate rises from

76.0% to 86.0%, but plateaus between depths 6 and 8 while tokens continue to escalate. Crucially, LeanRepair achieves **92.0% pass rate at 3,601 tokens/task**, establishing a strictly superior Pareto frontier: it outperforms the deepest static GoT search (J=8) by +6.0 pp while consuming fewer tokens than GoT at depth 6.

Panel B directly evaluates the reviewer-requested baselines: (i) GoT with a fixed attempt cap at J=5, and (ii) GoT with oscillation-triggered early stopping at attempt 5 (without LeanRepair's state guidance or recurrence buffer). The results yield two decisive conclusions: First, a static cap on GoT is brittle: on MBPP (Llama), imposing a fixed cap at J=5 causes pass rate to collapse from 65.0% to 47.5% (−17.5 pp), because many non-trivial synthesis tasks require multi-attempt iterative exploration. Adding oscillation-based stopping cushions this drop (preserving 62.5% pass rate). Second, trajectory stopping alone is insufficient: across all cells, full LeanRepair achieves strictly higher pass rates (e.g., 92.0% vs. 84.0% on Llama/QuixBugs; 96.0% vs. 84.0% on Qwen3/QuixBugs; 70.0% vs. 67.5% on Qwen3/MBPP) while simultaneously consuming fewer tokens. This demonstrates that LeanRepair does not merely "add early exit to GoT"; rather, edit-geometry guidance and recurrence prevention actively navigate the model out of local traps, while trajectory stopping prunes unproductive limit cycles.

C.8 Model-Scale Generalization

The cross-backbone study (Section 4.2) varies pretraining recipe at a roughly fixed parameter count (3.8–9B); it does not test whether LeanRepair's behavior is specific to that scale. We run all five methods (Self-Refine, CoT, ToT, GoT, and LeanRepair, under a matched eight-attempt budget) on three smaller members of the Qwen3 family (0.6B, 1.7B, and 4B) on the same four benchmarks and task counts as Table 1, alongside the already-reported Qwen3-8B row for reference. Section 4.3 and Figure 2 discuss the QuixBugs results. Figure 4 shows HumanEval, MBPP, and BigCodeBench, and Table 10 reports all four benchmarks. The Qwen3-14B row is complete for all five methods. At 14B, LeanRepair no longer leads on every benchmark: Self-Refine attains a higher BigCodeBench pass rate (80.0% vs. 73.3%) at roughly half the tokens, which is the direction the capacity–control hypothesis predicts as the backbone strengthens.

C.9 Quantization Robustness at 8B

To separate model scale from weight precision, we evaluate Qwen3-8B at three weight representations: FP16 (16-bit), bitsandbytes INT8, and bitsandbytes NF4 (4-bit). The benchmark task counts, eight-attempt budget, decoding settings, and five methods match Table 10. The FP16 rows reproduce the main-table Qwen3-8B results, including pruning for LeanRepair; the INT8 and NF4 LeanRepair rows disable pruning. Section 4.4 and Figure 2 discuss QuixBugs; Figure 5 shows the remaining benchmarks.

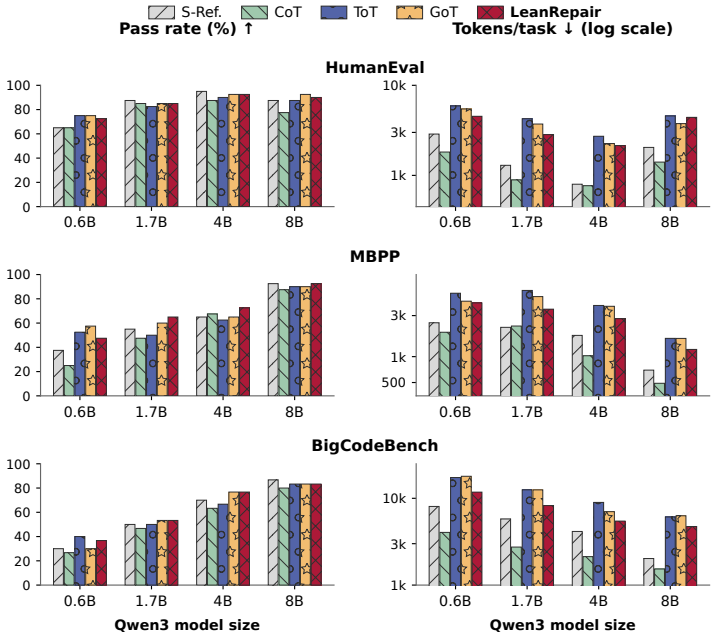


Figure 4: **Qwen3 model-scale comparison on the remaining benchmarks (Table 10).** QuixBugs appears in Figure 2. Rows show HumanEval, MBPP, and BigCodeBench; columns show pass rate (higher is better) and tokens/task (lower is better, logarithmic scale; k = 1,000). Grouped bars and hatch patterns identify the five methods, with LeanRepair in red (crosshatched). Token-axis ranges vary by benchmark and match Figure 5.

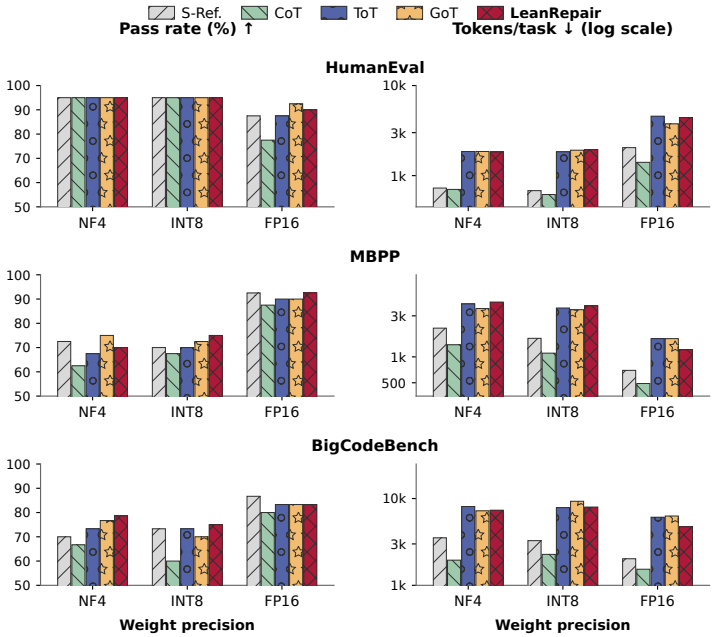


Figure 5: **Qwen3-8B precision comparison on the remaining benchmarks (Table 11).** QuixBugs appears in Figure 2. Rows show HumanEval, MBPP, and BigCodeBench; columns show pass rate (50–100%) and tokens/task (logarithmic scale; k = 1,000). Precision is ordered NF4, INT8, FP16. Method colors, hatch patterns, and benchmark-specific token-axis ranges match Figure 4.

Table 10: Pass rate and tokens/task across the Qwen3 model-scale family, grouped by scale. The four benchmarks, task counts, and method definitions for S-Ref., CoT, ToT, GoT, and LeanRepair match Table 1 (eight-attempt budget). Qwen3-1.7B, 4B, 8B, and 14B are complete for every method. Total Avg. is the mean of the four scale averages. Bold marks the best pass rate in each data or aggregate row.

		S-Ref.		CoT		ToT		GoT		**LeanRepair**	
Scale	Benchmark	Pass↑	Tok↓	Pass↑	Tok↓	Pass↑	Tok↓	Pass↑	Tok↓	Pass↑	Tok↓
Qwen3-1.7B	QuixBugs	60.0%	4,278	48.0%	2,345	48.0%	10,720	52.0%	10,651	**72.0%**	6,910
	HumanEval	**87.5%**	1,294	85.0%	891	82.5%	4,255	85.0%	3,717	85.0%	2,836
	MBPP	55.0%	2,191	47.5%	2,269	50.0%	5,884	60.0%	4,987	**65.0%**	3,563
	BigCodeBench	50.0%	5,778	46.7%	2,743	50.0%	12,524	**53.3%**	12,510	**53.3%**	8,244
	Avg	63.1%	3,385	56.8%	2,062	57.6%	8,346	62.6%	7,966	**68.8%**	5,388
Qwen3-4B	QuixBugs	68.0%	3,479	76.0%	1,400	60.0%	8,705	64.0%	7,857	**92.0%**	4,106
	HumanEval	**95.0%**	798	87.5%	768	90.0%	2,719	92.5%	2,256	92.5%	2,142
	MBPP	65.0%	1,766	67.5%	1,024	62.5%	3,935	65.0%	3,848	**72.5%**	2,764
	BigCodeBench	70.0%	4,151	63.3%	2,127	66.7%	8,971	**76.7%**	7,017	**76.7%**	5,465
	Avg	74.5%	2,549	73.6%	1,330	69.8%	6,083	74.6%	5,245	**83.4%**	3,619
Qwen3-8B	QuixBugs	76.0%	19,428	60.0%	17,979	56.0%	101,886	76.0%	71,842	**92.0%**	3,090
	HumanEval	87.5%	2,045	77.5%	1,405	87.5%	4,571	**92.5%**	3,757	90.0%	4,410
	MBPP	**92.5%**	697	87.5%	490	90.0%	1,631	90.0%	1,629	**92.5%**	1,213
	BigCodeBench	**86.7%**	2,023	80.0%	1,533	83.3%	6,095	**83.3%**	6,292	**83.3%**	4,734
	Avg	85.7%	6,048	76.3%	5,352	79.2%	28,546	85.5%	20,880	**89.5%**	3,362
Qwen3-14B	QuixBugs	**92.0%**	1,176	76.0%	1,210	84.0%	4,465	88.0%	3,587	**92.0%**	3,192
	HumanEval	92.5%	870	92.5%	738	**95.0%**	1,872	**95.0%**	1,962	**95.0%**	1,641
	MBPP	72.5%	1,713	70.0%	1,062	70.0%	3,575	72.5%	3,622	**77.5%**	2,957
	BigCodeBench	**80.0%**	2,416	63.3%	2,067	76.7%	6,317	76.7%	6,778	73.3%	4,648
	Avg	84.2%	1,543	75.5%	1,269	81.4%	4,057	83.0%	3,988	**84.5%**	3,110
Total Avg		76.9%	3,381	70.6%	2,503	72.0%	11,758	76.4%	9,520	**81.5%**	3,870

Table 11: Pass rate and tokens/task for Qwen3-8B under different weight quantizations, grouped by weight format. The FP16 block reproduces the main-table Qwen3-8B cells; NF4 and INT8 are separately evaluated with pruning disabled for LeanRepair. Each Avg. row averages the four benchmark rows within that weight format, while Total Avg. averages all 12 benchmark–format cells. Bold marks the best pass rate in each row, including ties.

		S-Ref.		CoT		ToT		GoT		**LeanRepair**	
Weight format	Benchmark	Pass↑	Tok↓	Pass↑	Tok↓	Pass↑	Tok↓	Pass↑	Tok↓	Pass↑	Tok↓
FP16 (16-bit)	QuixBugs	76.0%	19,428	60.0%	17,979	56.0%	101,886	76.0%	71,842	**92.0%**	3,090
	HumanEval	87.5%	2,045	77.5%	1,405	87.5%	4,571	**92.5%**	3,757	90.0%	4,410
	MBPP	**92.5%**	697	87.5%	490	90.0%	1,631	90.0%	1,629	**92.5%**	1,213
	BigCodeBench	**86.7%**	2,023	80.0%	1,533	83.3%	6,095	**83.3%**	6,292	**83.3%**	4,734
	Avg.	85.7%	6,048	76.3%	5,352	79.2%	28,546	85.5%	20,880	**89.5%**	3,362
INT8	QuixBugs	**96.0%**	1,688	72.0%	1,499	76.0%	5,517	84.0%	4,613	**96.0%**	3,358
	HumanEval	**95.0%**	681	**95.0%**	617	**95.0%**	1,839	**95.0%**	1,916	**95.0%**	1,946
	MBPP	70.0%	1,643	67.5%	1,105	70.0%	3,699	72.5%	3,536	**75.0%**	3,923
	BigCodeBench	73.3%	3,271	60.0%	2,274	73.3%	7,855	70.0%	9,301	**75.0%**	7,980
	Avg.	83.6%	1,821	73.6%	1,374	78.6%	4,728	80.4%	4,841	**85.2%**	4,302
NF4 (4-bit)	QuixBugs	84.0%	1,859	76.0%	1,230	80.0%	4,965	84.0%	4,416	**88.0%**	4,350
	HumanEval	**95.0%**	728	**95.0%**	703	**95.0%**	1,853	**95.0%**	1,854	**95.0%**	1,834
	MBPP	72.5%	2,158	62.5%	1,383	67.5%	4,128	**75.0%**	3,646	70.0%	4,337
	BigCodeBench	70.0%	3,525	66.7%	1,950	73.3%	8,097	76.7%	7,218	**78.6%**	7,342
	Avg.	80.4%	2,067	75.0%	1,316	79.0%	4,761	82.7%	4,283	**82.9%**	4,466
Total Avg.		83.2%	3,312	75.0%	2,681	78.9%	12,678	82.8%	10,002	**85.9%**	4,043