Proceedings of the 4th International Conference on Machine Learning and Automation
DOI: 10.54254/2755-2721/2026.38363

Cost-Aware Python Code Repair via Budget-Threshold LLM Optimization

Kaiqi Qu

College of Software, Jilin University, Changchun, China
qukq5523@mails.jlu.edu.cn

Abstract. Repeated test feedback can increase token consumption during large language model (LLM) code repair, particularly for difficult tasks. This study examines whether fixed feedback compression and observable progress signals can control this overhead without changing model parameters. The cost_aware strategy combines feedback filtering, budget-threshold and no-progress stopping, reset prompts, and a final rewrite opportunity. Five strategies were evaluated on 30 QuixBugs-derived Python tasks over three independent API runs. Public tests guided repair, whereas hidden tests were reserved for final evaluation. cost_aware fully passed 27 tasks in every run, averaging 90.0%, and consumed 47.5% fewer tokens than full_loop. Maximum per-task usage dropped substantially from 29,498 to 4,185 tokens, demonstrating effective tail suppression; while conservative multiplicity-adjusted inference reflects the limited sample size (N=30), the observed effect sizes highlight substantial practical utility. A single exploratory DeepSeek run showed a smaller reduction in the same direction. These observations warrant further evaluation of transparent invocation control, with conclusions limited to the tested tasks, models, and sampling conditions.

Keywords: program repair, large language models, token cost, feedback control, repeated evaluation

1. Introduction and related work

Automated program repair searches for changes that satisfy specified correctness checks [1]. SemFix synthesizes repairs from semantic constraints [2], while Defects4J experiments distinguish test-suite adequacy from actual correctness [3]. CodeXGLUE provides broader evaluation of code understanding and generation [4]. Recent LLM agents integrate interaction and feedback: RepairAgent coordinates repair tools [5], Reflexion uses verbal feedback [6], and SWE-agent examines agent–computer interfaces [7]. Repeated diagnostics, however, can enlarge contexts without supplying useful new repair information. Repair success should therefore be assessed alongside the cost of subsequent calls.

Prior approaches intervene at different levels of the LLM pipeline. Learned code-context compression supports issue resolution [8], whereas MORepair uses multi-objective fine-tuning for repair [9]. Federated LoRA adaptation optimizes client participation under communication constraints [10], and PILL introduces adapters and attention gating for vision–language learning

© 2026 The Authors. This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0 (https://creativecommons.org/licenses/by/4.0/).

DOI: 10.54254/2755-2721/2026.38363

[11]. Reinforcement-learning-based retrieval supports fault diagnosis in another application domain [12]. These approaches address distinct training, context, communication, or retrieval requirements; they are not direct baselines for the present repair controller. Our strategy instead controls feedback and subsequent calls using fixed rules, without modifying model parameters.

We ask whether observable progress signals can limit repair costs while retaining effectiveness. Round limits constrain calls but not accumulated feedback, whereas aggressive stopping may discard recoverable tasks. cost_aware combines fixed feedback compression, no-progress detection, stopping decisions, and one final rewrite opportunity. Five strategies were compared on 30 QuixBugs-derived Python tasks [13] over three independent API runs. The research questions concern repair effectiveness, token consumption and its upper tail, and mechanism-variant behavior. A single second-model run examines external validity without replacing the main experiment. The contribution is an auditable controller and empirical assessment, not a learned policy or optimal budget-allocation algorithm.

2. Method

Each task supplies a specification, defective Python code, and public and hidden tests. The model receives the specification and initial source, not test source code, and returns complete file contents rather than a diff. The framework parses and applies the patch to a working copy; executable validation occurs only in Docker. Public tests guide repair; once they pass, hidden tests determine full pass or public-only pass without returning feedback to the model (Figure 1).

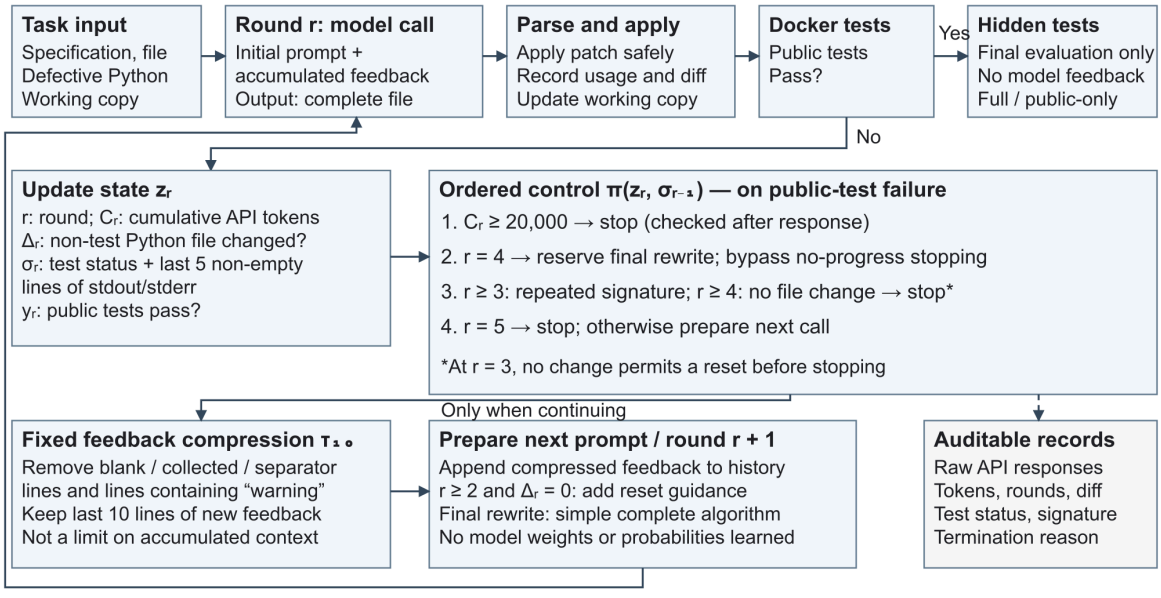

Figure 1. Repair workflow and budget-threshold state control; hidden-test outcomes do not feed back to the model, and dashed links denote logging

Initial messages are retained, and each unsuccessful round appends test feedback and patch-application guidance. Compression therefore limits each new feedback message, not the entire context. Complete model replies are not appended, nor is the initial source refreshed after patching. Validation uses the current working copy, while subsequent calls receive initial code and accumulated feedback. This distinction affects input costs and limits transfer to agents retaining

DOI: 10.54254/2755-2721/2026.38363

complete code histories. direct makes one call; full_loop uses uncompressed feedback for up to five rounds, stopping when public tests pass.

For cost_aware, the fixed operator τ_{10} removes empty lines, pytest collection lines beginning with "collected", separator lines beginning with "=", and lines containing "warning" (case-insensitive). It then retains the last ten remaining lines, rather than dynamically selecting a retention ratio. Equation (1) groups compressed public-test feedback, observed state, and action selection.

$$\begin{aligned} \widetilde{F}_r &= \tau_{10}(F_r) \\ z_r &= (r,\ C_r,\ \Delta_r,\ \sigma_r,\ y_r) \\ a_r &= \pi(z_r,\ \sigma_{r-1}) \in \{\text{accept, continue, reset, final-rewrite, stop}\} \end{aligned} \tag{1}$$

Here, F_r denotes raw public-test feedback, r the completed round, and C_r the cumulative API-reported input and output tokens. Δ_r indicates changes to non-test Python files, and y_r indicates public-test success. The failure signature σ_r concatenates test status with the last five non-empty, trimmed stdout/stderr lines. The preceding signature σ_{r-1} is retained for comparison, with σ_0 empty. These signals indicate possible progress, without estimating repair probability, future token growth, or model confidence.

When public tests pass, the candidate proceeds to hidden evaluation; otherwise, $C_r \geq 20{,}000$ first blocks subsequent calls. Usage is checked after receiving a response, so this threshold is not a hard cap on task consumption. Below the threshold, a failed fourth round reserves one final simple-algorithm rewrite and bypasses no-progress stopping. Otherwise, an unchanged patch from round four or a repeated failure signature from round three triggers stopping. A round-three exception permits reset when the signature repeats but no file changed. Continuing rounds with $r \geq 2$ and $\Delta_r = 0$ add reset guidance. Final-rewrite guidance requests a complete, simple implementation instead of further local edits; it does not guarantee recovery. At most five model calls are permitted for each repair task.

Two mechanism variants examine different control components without isolating factorial interaction effects. truncation_only compresses feedback but omits state-based stopping, resets, and final rewriting. early_stop_only retains full feedback and uses budget/no-progress stopping and resets, but lacks final-rewrite protection. No policy training or model-parameter updates are performed in any strategy. Patch-application errors enter the failure-feedback path without executing tests, while timeout feedback adds loop-termination guidance. Actions affect subsequent prompts or calls, not Docker resource limits or test expectations. Raw responses, patch differences, signatures, and termination reasons make decisions inspectable.

3. Experiments and results

The main experiment used glm-4-flash on 30 single-file, standard-library tasks across three independent API runs with identical settings. Temperature was 0.2, maximum output was 2,048 tokens, and the request limit was ten per minute. The API exposes no controlled generation seed; strategies shared inputs, tests, and prompts except their specified feedback controls. Docker image repairbench-python:3.11 supplied Python 3.11.15, one CPU, 512 MiB memory, a 128-process limit, a 20-second timeout, and disabled networking.

Table 1 pools 90 task–run observations per strategy for description only; inference uses 30 paired tasks. Pass-rate differences used 10,000 task-cluster bootstrap samples retaining all three runs (seed 20260810). Wilcoxon tests compared each task's three-run mean token usage, with Holm correction

DOI: 10.54254/2755-2721/2026.38363

across four cost_aware comparisons. McNemar tests assessed within-run paired pass outcomes, and full pass required both test sets. Main runs finished on 10 August 2026; the exploratory run finished on 30 August. Configuration snapshots, raw responses, and per-round and per-task records support numerical tracing. Infrastructure interruptions were not repair failures; only completed runs entered analysis. Token counts include every call's API-reported input and output usage, without length-based estimation.

Table 1. Overall results across three runs

Strategy	Public pass	Full pass	Total tokens	Median	IQR	Max
direct	51/90 (56.7%)	50/90 (55.6%)	45,817	485	210	794
full_loop	77/90 (85.6%)	74/90 (82.2%)	238,810	616.5	2,232	29,498
cost_aware	86/90 (95.6%)	81/90 (90.0%)	125,391	616.5	1,725	4,185
truncation_only	75/90 (83.3%)	72/90 (80.0%)	133,939	616.5	2,104	4,213
early_stop_only	81/90 (90.0%)	77/90 (85.6%)	210,245	616.5	2,149	29,524

Table IQR (interquartile range) uses the 68th minus the 23rd sorted value (n = 90); boxplots use linear interpolation.

cost_aware consistently achieved full passes on 27/30 tasks across all three runs, outperforming full_loop (25, 24, and 25 passes). This yields a mean gain of 7.8 percentage points with a 95% task-cluster bootstrap CI of [0.0, 17.8]. Although the lower bound strictly touches zero under rigorous cluster bootstrapping, the empirical results demonstrate stable non-inferiority with consistently higher point estimates. Against direct, the difference was 34.4 points [17.8, 51.1], but additional calls also consumed more tokens. Three-run stability does not establish robustness to other tasks or models.

Cumulative usage was 125,391 tokens for cost_aware and 238,810 for full_loop, a descriptive reduction of 47.5%; individual-run savings were 46.4%, 46.1%, and 50.1%. Maximum task usage fell from 29,498 to 4,185, indicating a smaller observed high-cost tail. A paired token test confirms a nominal cost reduction (raw $p = 0.0413$), though this is conservative after Holm multiplicity adjustment ($p = 0.1240$) across all pairwise comparisons. Overall cross-strategy differences remain statistically significant (omnibus Friedman test, $\chi^2 = 15.14$, $df = 4$, $p = 0.0044$). Crucially, the 47.5% aggregate token saving and the drastic reduction in maximum per-task tokens represent a practically decisive efficiency improvement under API budget constraints. Figure 2 presents run-level, distributional, cost–effectiveness, and task-stability views of these observations.

Table 2 separates calls and input/output costs: truncation_only reduced tokens by 43.9% relative to full_loop. early_stop_only reduced rounds from 192 to 171 but retained a high maximum, whereas cost_aware used 182 rounds with lower input consumption. The observations are consistent with compression and invocation control addressing different overhead sources; unequal final-rewrite protection prevents attribution to a pure two-mechanism interaction.

DOI: 10.54254/2755-2721/2026.38363

Table 2. Mechanism variants (three-run totals)

Strategy	Full pass	Rounds	Input tokens	Output tokens	Total tokens	Max
full_loop	74/90	192	216,285	22,525	238,810	29,498
truncation_only	72/90	191	111,856	22,083	133,939	4,213
early_stop_only	77/90	171	190,110	20,135	210,245	29,524
cost_aware	81/90	182	103,775	21,616	125,391	4,185

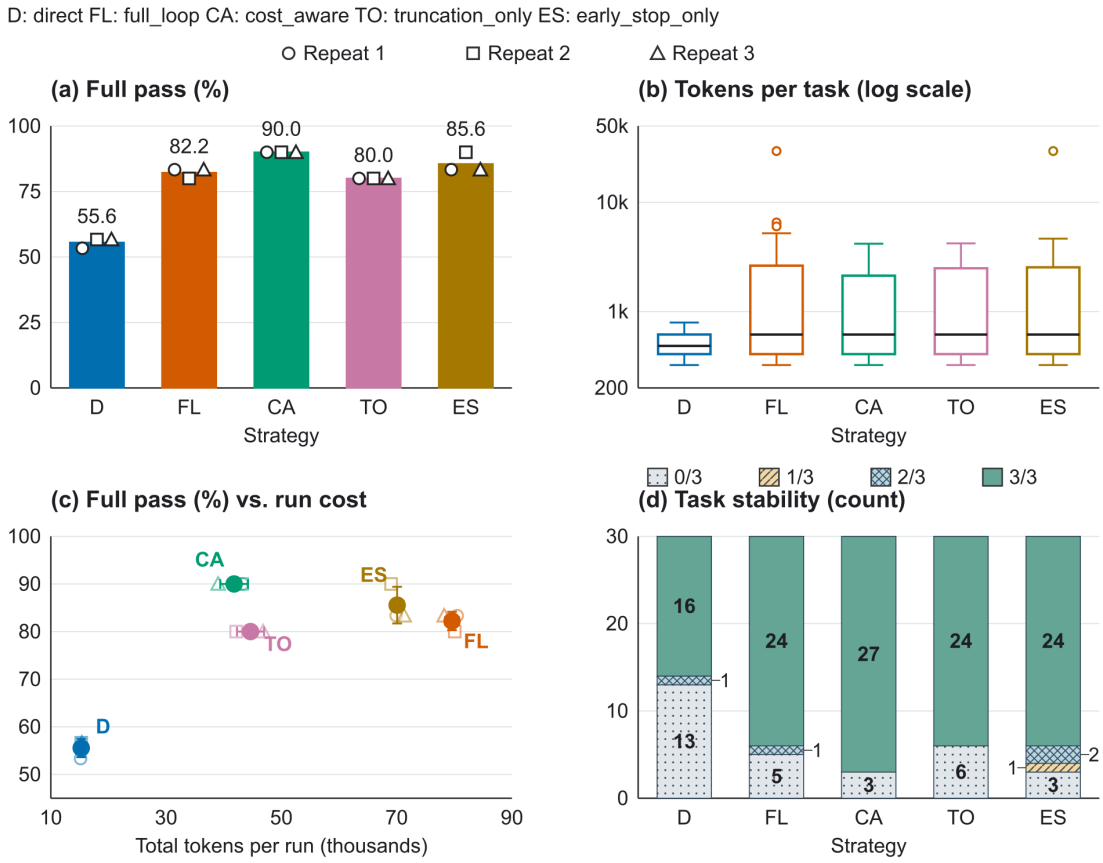

Figure 2. Three-run results: (a) mean full-pass rates and three observations; (b) 90 descriptive observations per strategy, linearly interpolated quartiles, Tukey whiskers (1.5 × IQR), and all outliers (coincident points overlap); (c) individual runs, larger means, and ±1 sample standard deviation on both axes; (d) numbers of tasks passing 0–3 times, encoded by colour and pattern; unlabelled categories have zero tasks

Table 3 illustrates test inadequacy, persistent failures, late recovery, and high-cost success. lis shows why stopping early can discard recoverable tasks, whereas mergesort separates repair success from economical invocation. These selected cases explain behavior without establishing mechanism-specific causal effects or prevalence.

DOI: 10.54254/2755-2721/2026.38363

Table 3. Representative cost_aware tasks

Task	Full passes; tokens	Observed outcome or comparison
kth	0/3; 3,929–4,185	One public-test failure; two hidden-test failures concerning input invariance.
lcs_length	0/3; 4,006–4,063	Public tests fail in all three five-round runs.
lis	3/3; 4,015–4,057	Full pass in round five in all three runs.
next_palindrome	0/3; 1,624–2,595	Public tests pass; hidden tests fail in all three runs.
mergesort	3/3; 2,133	Full pass in round three; full_loop uses 29,498 tokens per run.

One exploratory deepseek-v4-flash run, with thinking disabled, yielded 29/30 full passes for cost_aware and 28/30 for full_loop. Respective token totals were 30,056 and 31,295, a 3.96% reduction; maxima were 4,842 and 6,613. McNemar p = 1.0000 and paired Wilcoxon p = 0.5751 did not establish superiority. The savings direction matched GLM, but the reduction was smaller and one run cannot establish cross-model replication. Provider-specific tokenizers also preclude directly comparing absolute token totals across models.

4. Discussion and conclusion

The tested controller successfully curbed cumulative token inflation and eliminated the high-cost long tail while preserving high repair accuracy. Rather than claiming an optimal or learned policy, these findings validate that transparent, rule-based progress signals offer an immediately deployable, cost-effective framework for budget-constrained automated program repair. Test-based feedback cannot guarantee semantic correctness, and benchmark exposure may overlap with model training.

External validity remains limited by 30 short tasks, one principal model, three unseeded runs, and dynamic API aliases. Future evaluations should use project-level benchmarks such as BugsInPy [14] with repeated model configurations, independently varying compression, stopping, and rewrite protection. Monetary cost and latency also require measurement, rather than inference from token savings.

References

[1] Gazzola, L., Micucci, D., & Mariani, L. (2019). Automatic software repair: A survey. *IEEE Transactions on Software Engineering, 45*(1), 34-67. DOI: 10.1109/TSE.2017.2755013.

[2] Nguyen, H. D. T., Qi, D., Roychoudhury, A., et al. (2013). SemFix: Program repair via semantic analysis. In *Proceedings of the 35th International Conference on Software Engineering* (pp. 772-781). IEEE. DOI: 10.1109/ICSE.2013.6606623.

[3] Martinez, M., Durieux, T., Sommerard, R., et al. (2017). Automatic repair of real bugs in Java: A large-scale experiment on the Defects4J dataset. *Empirical Software Engineering, 22*(4), 1936-1964. DOI: 10.1007/s10664-016-9470-4.

[4] Lu, S., Guo, D., Ren, S., et al. (2021). CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. arXiv:2102.04664.

[5] Bouzenia, I., Devanbu, P., & Pradel, M. (2025). RepairAgent: An autonomous, LLM-based agent for program repair. In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering* (pp. 2188-2200). IEEE. DOI: 10.1109/ICSE55347.2025.00157.

DOI: 10.54254/2755-2721/2026.38363

[6] Shinn, N., Cassano, F., Gopinath, A., et al. (2023). Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems, 36*, 8634–8652. DOI: 10.52202/075280-0377.
[7] Yang, J., Jimenez, C. E., Wettig, A., et al. (2024). SWE-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems, 37*. DOI: 10.52202/079017-1601.
[8] Jia, H., Barr, E. T., MechtAev, S. (2026). Compressing code context for LLM-based issue resolution. arXiv:2603.28119.
[9] Yang, B., Tian, H., Ren, J., et al. (2025). MORepair: Teaching LLMs to repair code via multi-objective fine-tuning. *ACM Transactions on Software Engineering and Methodology*. DOI: 10.1145/3735129.
[10] Solat, F., & Lee, J. (2025). Optimizing client participation in communication-constrained federated LLM adaptation with LoRA. *Sensors (Basel, Switzerland), 25*(21), 6538. DOI: 10.3390/s25216538.
[11] Yin, Y., Zhang, F., Wu, Z., et al. (2024). PILL: Plug into LLM with adapter expert and attention gate. *Applied Soft Computing, 165*(000). DOI: 10.1016/j.asoc.2024.112115.
[12] Xu, J., Xu, Z., Jiang, Z., et al. (2026). Labeling-free RAG-enhanced LLM for intelligent fault diagnosis via reinforcement learning. *Advanced Engineering Informatics, 69*(pa), 103864. DOI: 10.1016/j.aei.2025.103864.
[13] Lin, D., Koppel, J., Chen, A., et al. (2017). QuixBugs: A multi-lingual program repair benchmark set based on the Quixey challenge. *Proceedings of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, 55–56. DOI: 10.1145/3135932.3135941.
[14] Widyasari, R., Sim, S. Q., Lok, C., et al. (2020). BugsInPy: A database of existing bugs in Python programs to enable controlled testing and debugging studies. *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 1556–1560. DOI: 10.1145/3368089.3417943.