
Frozen Inner Looping Is Brittle for Code Generation with BitNet b1.58

Suhyeong Shin¹ Sujeong Ji¹ Yeongwook Yang¹

¹Department of Computer Engineering, Kangwon National University

shin0815@kangwon.ac.kr, jisujeong25@kangwon.ac.kr, yeongwook.yang@kangwon.ac.kr

Abstract

We show that frozen inner looping, repeating a selected decoder-block range T times inside a frozen model, is not a reliable accuracy knob for native 1.58-bit code generation. We test BitNet b1.58 2B4T with a source-preserving schedule patch, using ordinary one-pass inference ($T = 1$) as a paired baseline on 742 tasks from HumanEval+, MBPP+, BigCodeBench-Instruct, and DS-1000. We compare two simple reinjection rules: a running average of repeated block outputs and a damped residual blend. Running-average reinjection is flat or worse on the main EvalPlus suites. Damped residual reinjection at $T = 2$ changes 30 failures to passes and 19 passes to failures, increasing raw pass count from 207/742 to 218/742, but the aggregate effect is not statistically confirmed (paired exact $p = 0.152$, 95% bootstrap CI $[-0.27, +3.37]$ percentage points). At $T = 8$, performance collapses to 66/742. Output comparisons show large divergence from $T = 1$ and asymmetric pass-to-fail transitions at deeper repetition, suggesting perturbation rather than productive refinement.

1. Introduction

Local code generation has a concrete deployment bottleneck: stronger coding agents often rely on large models or external test-time loops, while private or resource-constrained use is limited by memory and latency. Native 1.58-bit LLMs such as BitNet b1.58 2B4T promise a different operating point because the backbone is trained under low-bit weight constraints rather than compressed after training (Ma et al., 2025). This makes inner looping a targeted stress test: can extra hidden computation improve one greedy completion, or does repeating frozen decoder blocks perturb the hidden trajectory expected by later layers?

Proceedings of the 43rd International Conference on Machine Learning, Seoul, South Korea. PMLR 306, 2026. Copyright 2026 by the author(s).

The answer matters for code because a single missing condition, wrong entry point, or malformed literal can turn a plausible solution into an execution failure. Benchmarks such as HumanEval+, MBPP+, BigCodeBench, and DS-1000 therefore expose small semantic changes that text metrics would miss (Chen et al., 2021; Austin et al., 2021; Liu et al., 2023; EvalPlus Team, 2024; Zhuo et al., 2025; Lai et al., 2023). Layer repetition could refine the trajectory induced by low-bit weights, but it could also push representations off the distribution expected by downstream layers.

Because released low-bit models are most attractive when retraining or large external search is costly, post-release inference-time computation is a natural intervention. Many coding-agent methods spend extra compute through sampling, reranking, execution-grounded selection, or repair (Chen et al., 2021; Li et al., 2025; Olausson et al., 2024). Inner loop inference is different: it repeatedly applies a selected range of Transformer blocks inside the model, without observing test results and without changing weights (Lys et al., 2026). In a native 1.58-bit model, this creates a sharper mechanistic question. Repetition could smooth or refine a noisy representation, but it could also compound approximation error or produce hidden states that later layers were never trained to consume.

We therefore ask:

What happens when selected decoder layers are repeated at test time inside a native 1.58-bit code-capable LLM?

We answer with a controlled, BitNet-specific empirical study. Across two simple reinjection rules introduced below, running-average looping does not yield a robust gain on HumanEval+ or MBPP+, while a damped residual loop increases raw pass counts on each evaluated suite or slice at $T = 2$. The aggregate effect remains modest and statistically unconfirmed, and larger loop counts cause severe degradation. This pattern suggests that untrained repetition is better understood as a brittle hidden-state intervention than as reliable “more thinking.”

Our contributions are a forward-preserving BitNet inner-

loop implementation, a 742-task paired execution evaluation across four code benchmarks, and evidence that frozen looping is brittle: $T = 2$ residual updating gives a small, statistically unconfirmed positive shift, $T = 4$ is weaker, and $T = 8$ collapses.

2. Related Work

Execution-based code evaluation. Code-generation papers increasingly rely on execution rather than surface-form similarity. HumanEval and MBPP evaluate short Python programs (Chen et al., 2021; Austin et al., 2021). EvalPlus augments HumanEval with substantially stronger tests to form HumanEval+ (Liu et al., 2023), and the EvalPlus project also provides MBPP+ as an augmented MBPP benchmark (EvalPlus Team, 2024). BigCodeBench broadens evaluation to diverse function-call and tool-use composition (Zhuo et al., 2025), while DS-1000 targets data-science code across Python libraries (Lai et al., 2023). We likewise treat pass/fail execution as the outcome.

Test-time computation for code. Sampling multiple candidates has long been used in code generation (Chen et al., 2021); selection and coverage are central to recent test-time scaling work (Li et al., 2025); and repair methods use compiler or execution feedback to revise generated programs (Olausson et al., 2024). Our setting removes these mechanisms: it produces one greedy completion and spends the additional compute inside the frozen hidden trajectory before any execution result is observed.

Native low-bit models and inner recurrence. Low-bit LLM work is usually motivated by memory, energy, and latency. BitNet b1.58 trains ternary weights natively rather than quantizing a full-precision model (Ma et al., 2025). Inner loop inference instead asks whether repeated blocks can unlock latent capabilities without training (Lys et al., 2026). In a native 1.58-bit-weight model, this is also a perturbation of the representation stream induced by that backbone. We study that intersection for executable code generation.

3. Method

Intervention boundary. We study one frozen test-time intervention: a change to the decoder-layer schedule inside BitNet, applied before any candidate program is executed. Let the decoder contain layers $F_0, \dots, F_{L-1}$ acting on residual stream h . The ordinary schedule is $(0, 1, \dots, L-1)$. For a contiguous range $[s, e)$ and loop count T , inner looping uses

$$(0, \dots, s-1, \underbrace{s, \dots, e-1, \dots, s, \dots, e-1}_{T \text{ passes}}, e, \dots, L-1).$$

No weights, prompts, decoding rules, postprocessors, rerankers, or execution feedback are changed. The only intervention is whether BitNet’s residual trajectory revisits a selected decoder substack.

Source-preserving implementation. We patch the BitNet/Transformers implementation recommended by the Microsoft model card (Microsoft, 2025; Wolf et al., 2020), replacing only the loop over decoder layers. The patched forward keeps attention masks, position IDs, cache positions, rotary embeddings, attention flags, return structure, and extra keyword arguments unchanged. All conditions disable KV caching because within-token reuse of the same layer makes cache semantics ill-defined; this keeps baseline and looped runs comparable.

Update rules. For each relative position r in the repeated substack, let $j = s + r$ and let $y_{t,r} = F_j(h)$ be the ordinary BitNet block output on visit t . We evaluate two update rules for reinjecting repeated block outputs into the residual stream. The running-average rule maintains a per-forward mean for each relative layer position:

$$\begin{aligned} m_{0,r} &= 0, \\ m_{t,r} &= \frac{(t-1)m_{t-1,r} + y_{t,r}}{t}, \\ h &\leftarrow m_{t,r}, \end{aligned}$$

where the average is local to position r and never mixes different decoder layers. The residual rule instead interpolates toward the current layer output y :

$$h \leftarrow (1 - \alpha)h + \alpha y.$$

Following inner-loop inference work that repeats internal Transformer block ranges (Lys et al., 2026), we use the pre-specified mid-stack window $[12, 16)$ in the 30-layer BitNet backbone as a compact stress-test range; we do not claim this range is inherited from prior work or optimal. The $T = 1$ condition denotes ordinary one-pass inference. For residual updates, we fix $\alpha = 0.5$ as a symmetric equal-blend damping coefficient, fixed before aggregate evaluation rather than tuned to benchmark outcomes.

Equivalence checks. The paper hinges on the schedule change, so the cache-disabled no-extra-loop path must match the unmodified model. Before benchmark evaluation, logits must match within 10^{-5} , greedy next-token choices must match, and full greedy generations must match on three fixed prompts. For looped settings, we require finite logits and unchanged tensor shapes/return types for $T \in \{2, 4, 8\}$. No completion is manually edited.

Table 1. Pass counts for residual inner looping with $\alpha = 0.5$ on layers [12, 16]. Values are passed/total; Aggregate pools the same 742 paired tasks.

T	HumanEval+	MBPP+	BCB	DS-1000	Aggregate
1	49/164	141/378	11/100	6/100	207/742
2	51/164	147/378	13/100	7/100	218/742
4	50/164	126/378	8/100	5/100	189/742
8	16/164	47/378	1/100	2/100	66/742

4. Experiments

Setup. We evaluate BitNet b1.58 2B4T, released as microsoft/bitnet-b1.58-2B-4T, using a pinned Transformers implementation. Each condition produces one Python-only greedy completion per task from the same chat-template prompt, with caching disabled consistently across baseline and looped runs. We use no sampling, reranking, repair, or execution feedback. Completions are sanitized or extracted deterministically, then scored by each benchmark’s execution tests; parsing failures, exceptions, timeouts, truncation, and failed tests count as failures.

Benchmarks. We use 742 paired tasks: HumanEval+ and MBPP+ via EvalPlus passed_plus (Chen et al., 2021; Austin et al., 2021; Liu et al., 2023; EvalPlus Team, 2024), plus fixed 100-task slices from BigCodeBench-Instruct (Zhao et al., 2025) and DS-1000 (Lai et al., 2023), both selected before observing loop outcomes. BigCodeBench-Instruct probes library/tool-call composition; DS-1000 spans seven Python libraries, five covered by our cost-limited slice.

Loop settings and analysis. Under one paired exploratory design, we keep the pre-specified range [12, 16]. Residual reinjection uses a doubling schedule, $T \in \{1, 2, 4, 8\}$, from the unlooped baseline to a deeper stress test; running-average reinjection is evaluated at the matched moderate-depth setting, $T = 4$. All comparisons are paired by task against $T = 1$. We report pass counts, fail-to-pass/pass-to-fail transitions, exact sign-test p -values, and paired bootstrap CIs.

5. Results

The running-average branch is flat or worse. The running-average configuration, $T = 4$ over layers [12, 16], is nearly flat on HumanEval+ (50/164 vs. 49/164 for $T = 1$, $p = 1.0$) and slightly worse on MBPP+ (139/378 vs. 141/378, $p = 0.754$). Its transitions look more like semantic perturbations than refinement: some tasks flip from fail to pass, but a comparable number flip from pass to fail.

Residual updating gives a small, unconfirmed $T = 2$ signal. Table 1 shows the main T-curve for residual updating

Table 2. Paired aggregate transitions for residual inner looping relative to $T = 1$. Delta is pass-rate difference in percentage points; p is the exact sign-test value over discordant pairs.

Target	Delta	F→P	P→F	p
$T = 2$	+1.48 pp	30	19	0.152
$T = 4$	-2.43 pp	42	60	0.092
$T = 8$	-19.00 pp	9	150	4.1×10^{-34}

ing with $\alpha = 0.5$. At $T = 2$, residual updating has higher raw pass counts on every suite: +2 HumanEval+ tasks, +6 MBPP+ tasks, +2 BigCodeBench-Instruct tasks, and +1 DS-1000 task. Aggregated over 742 paired tasks, this is an 11-task, +1.48 percentage-point difference.

However, the evidence is not strong enough for a confirmed improvement claim. Table 2 shows that the paired aggregate comparison has 30 fail-to-pass transitions and 19 pass-to-fail transitions, with exact $p = 0.152$ and 95% bootstrap CI $[-0.27, +3.37]$ percentage points. Individual benchmark CIs also include zero.

Larger loop counts collapse. The clearest statistically supported effect is negative: larger loop counts collapse performance. At $T = 8$, pass count falls from 207/742 to 66/742. The transition matrix is highly asymmetric: only 9 tasks improve while 150 regress. This is consistent with repeated reuse of frozen decoder blocks moving representations away from the distribution expected by later layers; the following output comparisons support this view. The residual update may limit this shift at $T = 2$, but it does not make repeated application safe.

Changed outputs resemble perturbations, not stable repairs. Although we do not directly measure hidden-state drift, exact string comparisons provide output-level evidence of accumulating perturbation. Relative to $T = 1$, sanitized solutions differ on 57.3%, 83.2%, and 96.1% of tasks at $T = 2, 4, 8$, while pass/fail outcomes change on only 6.6%, 13.7%, and 21.4%. The gap indicates that most perturbations are neutral under the benchmark execution tests; however, when correctness changes, the asymmetry sharpens, reaching 150 pass-to-fail versus 9 fail-to-pass transitions at $T = 8$. This pattern is consistent with accumulating off-distribution perturbation rather than stable refinement.

6. Discussion

Why code exposes the limitation. Inner looping changes hidden states before generation but receives no execution feedback. Code magnifies this risk: a one-token branch or entry-point error can invalidate an otherwise plausible solution. The output-divergence and transition patterns above suggest that frozen repetition mostly moves decisions near

the model’s existing boundary rather than providing a stable repair process.

Frozen recurrence is not trained recurrence. These results do not contradict latent-recurrence work. Prior positive results include arXiv preprint work reporting inference-time inner looping in off-the-shelf pretrained models, where repeated block ranges yield modest improvements and stable latent trajectories (Lys et al., 2026), and architectures trained to reuse a recurrent block at variable depth (Geiping et al., 2025). Our intervention asks a harsher question: can an off-the-shelf native 1.58-bit code model tolerate repeated decoder blocks that were not trained for repeated application? For this model and window, the answer is mostly no; residual updating at $T = 2$ is a candidate follow-up setting, not a confirmed improvement. For low-bit coding agents, extra hidden compute is not automatically useful unless the model or selection process controls it.

Limitations. This study uses one native 1.58-bit model, one primary block range, and no no-op repeat control, so the result is BitNet-specific evidence rather than a universal failure of frozen recurrence; schedule reuse and reinjection effects are not fully separated. Because update rule, loop depth, α , and block range were fixed in one exploratory design, the $T = 2$ point is not a multiplicity-corrected discovery. BigCodeBench and DS-1000 use fixed 100-task slices for cost reasons, not full-benchmark estimates. Our cache-disabled path evaluates correctness effects, not specialized low-bit kernels.

Reproducibility. We release per-task CSVs with completions, sanitized solutions, pass/fail labels, latency/token metadata, prompt/postprocessing modes, loop settings, and source diffs. EvalPlus uses version 0.3.1, with loader-returned counts of 164 HumanEval+ and 378 MBPP+ tasks. The model artifact is microsoft/bitnet-b1.58-2B-4T; the pinned Transformers fork uses the model card’s recommended commit 096f25a (Microsoft, 2025; Wolf et al., 2020).

7. Conclusion

Frozen test-time inner looping is not a reliable accuracy knob for BitNet b1.58 2B4T code generation. In a 742-task paired evaluation, running-average updating is flat or worse on the EvalPlus suites; damped residual updating at $T = 2$ yields higher raw pass counts but remains statistically unconfirmed; and deeper repetition sharply degrades. The practical boundary in this setting is clear: native low-bit coding agents are sensitive to hidden-state reuse, and reliable gains likely require recurrence trained for reuse, feedback-aware selection and repair, or broader model/range controls that show when reuse is safe.

References

- Austin, J. et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.
- Chen, M. et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.
- EvalPlus Team. EvalPlus v0.3.1. Software repository, 2024. Released 2024-10-20; accessed 2026-05-11.
- Geiping, J. et al. Scaling up test-time compute with latent reasoning: A recurrent depth approach. In *Advances in Neural Information Processing Systems*, 2025. Spotlight.
- Lai, Y. et al. DS-1000: A natural and reliable benchmark for data science code generation. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 18319–18345. PMLR, 2023.
- Li, D. et al. S^* : Test time scaling for code generation. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pp. 15964–15978, 2025. doi: 10.18653/v1/2025.findings-emnlp.865.
- Liu, J., Xia, C. S., Wang, Y., and Zhang, L. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems*, 2023.
- Lys, J. et al. Inner loop inference for pretrained transformers: Unlocking latent capabilities without training. *arXiv preprint arXiv:2602.14759*, 2026.
- Ma, S. et al. BitNet b1.58 2B4T Technical Report. *arXiv preprint arXiv:2504.12285*, 2025.
- Microsoft. BitNet b1.58 2B4T - Scaling Native 1-bit LLM. Hugging Face model card, 2025. Accessed 2026-05-11.
- Olausson, T. X. et al. Is self-repair a silver bullet for code generation? In *The Twelfth International Conference on Learning Representations*, 2024.
- Wolf, T. et al. Transformers: State-of-the-art natural language processing. In *EMNLP System Demonstrations*, 2020.
- Zhuo, T. Y. et al. BigCodeBench: Benchmarking code generation with diverse function calls and complex instructions. In *The Thirteenth International Conference on Learning Representations*, 2025.