BEYOND BLIND GENERATION SCALING: VERIFICATION-GUIDED INFERENCE FOR DIFFUSION LANGUAGE MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Test-time scaling typically improves language-model performance by spending additional inference compute on generation, implicitly assuming that more generation compute directly yields more reliable outputs. We show that this view is incomplete for diffusion language models. By decomposing model outcomes according to their observability, we identify a compute-dependent loud-to-silent-to-correct transition in the output distribution: as denoising compute increases, loud failures decline before semantic errors disappear, creating an intermediate regime in which surface-valid but incorrect outputs become dominant among the remaining failures before correctness eventually catches up. This reveals that surface validity can improve before semantic correctness. Motivated by this failure transition, we introduce *verification-guided inference*, a new test-time inference paradigm that moves beyond blind generation scaling by separating candidate generation from output assurance. The model first generates with a moderate budget, filters directly observable validity failures with a lightweight deterministic check, and applies an independent verifier to the remaining candidates; only rejected candidates receive higher-budget regeneration. Across code generation and mathematical reasoning, the transition persists across diffusion model families and evaluation controls. On security-sensitive programming tasks, silent failures can additionally include functionally valid yet vulnerable programs. Independent verification detects 72–83% of silent failures with a single pass costing only 2.3% of the token-forward compute of 256-step generation. Under full coverage, verification-guided inference improves correctness from 41.1% to 48.0%, reduces silent failures from 45.8% to 34.0%, and lowers average generation-equivalent compute from 256 to 243.

1 INTRODUCTION

Test-time scaling has become an increasingly important way to improve language-model performance without modifying model parameters. With additional inference compute, a model can reason for more steps (Wei et al., 2022), sample more candidates (Wang et al., 2022; Brown et al., 2024), refine previous responses (Madaan et al., 2023), or, in diffusion language models (Nie et al., 2025; Ye et al., 2025), perform more denoising steps before producing its final output. Despite these different mechanisms, existing scaling strategies largely share the same operational assumption: more inference compute should be spent on generation because more generation is expected to produce better answers. This assumption appears well supported when performance is summarized by accuracy, since larger generation budgets often increase the probability of obtaining a correct solution. Accuracy, however, says little about what happens to the failures that remain. A program that cannot be parsed or executed and a program that executes normally but silently produces an incorrect result are both counted as wrong, even though they differ substantially in how easily a downstream system can recognize and recover from them. This distinction motivates a more fundamental question for test-time scaling:

what does additional generation compute actually buy before an incorrect output becomes correct?

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.

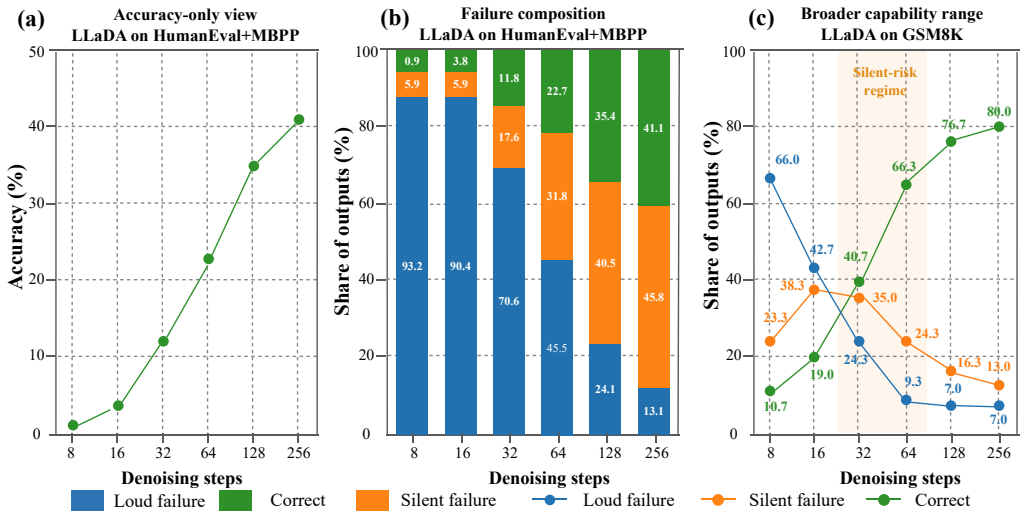


Figure 1: **Failure-mode evolution under diffusion test-time scaling.** (**a**) Accuracy of LLaDA on HumanEval and MBPP across denoising budgets. (**b**) The corresponding output distribution, decomposed into correct outputs, *loud failures* (invalid or non-executable), and *silent failures* (valid but semantically incorrect). (**c**) The same decomposition on GSM8K, where the remaining failures shift from loud toward silent as generation compute increases, before correctness becomes dominant at higher budgets.

We study this question in diffusion language models, where generation compute can be controlled directly through the number of denoising steps. At first glance, the conventional view appears to hold. As the generation budget increases, task accuracy improves steadily, producing the familiar scaling behavior shown in Figure 1(a). The picture changes, however, once the remaining failures are distinguished by their observability. We refer to outputs whose invalidity can be immediately exposed through parsing, execution, or output-format checks as *loud failures*, and to outputs that satisfy these basic validity requirements but remain semantically incorrect as *silent failures*. Under this decomposition, increasing generation compute changes not only the overall error rate but also the distribution of failure modes. On LLaDA (Nie et al., 2025) over HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021b), increasing the denoising budget from 8 to 256 steps raises correctness from 0.9% to 41.1%, while loud failures fall from 93.2% to 13.1% and silent failures rise from 5.9% to 45.8%. Figure 1(b) therefore reveals that the scaling trajectory does not move directly from an error-dominated regime to a correct-output regime. Instead, loud failures disappear substantially faster than semantic errors, shifting the remaining error distribution toward surface-valid but incorrect outputs. This shift does not continue indefinitely. On GSM8K, silent failures first increase as loud failures disappear, but later decline as correctness continues to improve. Figure 1(c) therefore completes the scaling trajectory: low-compute outputs are dominated by loud failures, intermediate-compute failures increasingly shift toward the silent category, and higher compute increasingly yields correct outputs. The central phenomenon is that *surface validity improves before semantic correctness*. Silent failure is therefore not simply an error type that grows with compute, but an intermediate regime that emerges while outputs have become well formed and apparently usable but semantic correctness has not yet caught up.

We find that the failure-composition shift persists across diffusion model families, stronger test suites, prompt variations, and decoding controls. The intermediate silent regime is also operationally meaningful: on security-sensitive code-generation tasks, some outputs pass functionality tests while remaining vulnerable under dedicated security oracles. Thus, improved surface validity does not necessarily imply improved reliability. This failure transition exposes a limitation of *blind generation scaling*. At intermediate budgets, candidates already occupy different reliability states: some remain overtly invalid, some are surface-valid but semantically unreliable, and some are already reliable enough to return. Uniformly applying the largest generation budget ignores this information. We therefore ask a different inference question: once an intermediate candidate is available, should

additional compute still be spent uniformly on generation, or should the candidate itself determine whether more generation is necessary?

Motivated by this observation, we introduce *verification-guided inference*, a new test-time inference paradigm that separates candidate generation from output assurance. The model first generates with a moderate budget, directly observable failures are filtered using a lightweight deterministic validity check, and the remaining candidates are evaluated by an independent verifier. Only candidates rejected by either assurance stage receive higher-budget regeneration. The verifier is therefore not used to predict whether additional generation will necessarily repair a candidate; instead, it determines whether the current candidate is reliable enough to terminate inference. Our experiments show that independent verification provides a substantially stronger reliability signal than self-verification at low additional cost. Combined with selective regeneration, verification-guided inference improves end-to-end reliability while reducing average inference compute compared with blind high-budget generation. Our contributions are threefold:

- We identify a compute-dependent failure-distribution transition in diffusion language models, in which the dominant regime shifts from loud failures to silent failures before correctness becomes dominant, revealing that surface validity can improve before semantic correctness.
- We show that this transition persists across diffusion model families and evaluation controls and can produce operationally important silent failures, including functionally valid but vulnerable programs.
- We introduce *verification-guided inference*, a new test-time inference paradigm that conditions higher-budget generation on the observed candidate through deterministic validity checking and independent semantic verification, improving the end-to-end reliability–compute trade-off.

2 Related Work

Test-Time Scaling. Test-time scaling improves model performance through longer reasoning (Wei et al., 2022), repeated sampling (Wang et al., 2022; Brown et al., 2024), refinement (Madaan et al., 2023), search, or adaptive stopping. Recent work further studies how inference budgets should vary with task difficulty, uncertainty, or expected utility (Snell et al., 2024). These approaches are typically evaluated through accuracy, reward, or pass rate, collapsing different failure modes into a single incorrect outcome. We instead study how the *distribution and observability* of model outcomes change with generation compute, showing that the dominant regime can shift from loud failures to silent failures before correctness becomes dominant.

Diffusion Language Models. Diffusion language models generate text through iterative denoising (Austin et al., 2021a; Lou et al., 2023; Sahoo et al., 2024), making the number of denoising steps a direct test-time compute axis (Nie et al., 2025; Ye et al., 2025). Prior work has focused on generation quality and reasoning capability (Nie et al., 2025; Ye et al., 2025; Xie et al., 2025), decoding efficiency and sampling schedules (Li et al., 2026; Sedykh et al., 2026), and reducing the number of denoising steps needed to reach a target level of performance (Rottoli et al., 2026; Dou & Shu, 2026). We instead examine how output composition evolves along this compute axis, revealing an intermediate regime in which surface validity improves faster than semantic correctness and silent failures become increasingly prominent.

Verification and Self-Correction. Verification and self-correction are widely used for candidate scoring and reranking (Cobbe et al., 2021; Lightman et al., 2024), critique, search guidance, and iterative revision (Madaan et al., 2023). Verifiers may be implemented through the generator itself (Madaan et al., 2023), a separate model (Cobbe et al., 2021; Lightman et al., 2024), or an external signal, while self-correction without external feedback can remain unreliable (Huang et al., 2024). Existing systems mainly use verification to select, rerank, or revise generated candidates. We instead make verification part of the test-time inference policy: a moderate-budget candidate first undergoes deterministic validity checking and independent semantic verification, and higher-budget generation is invoked only when the current candidate is not reliable enough to return.

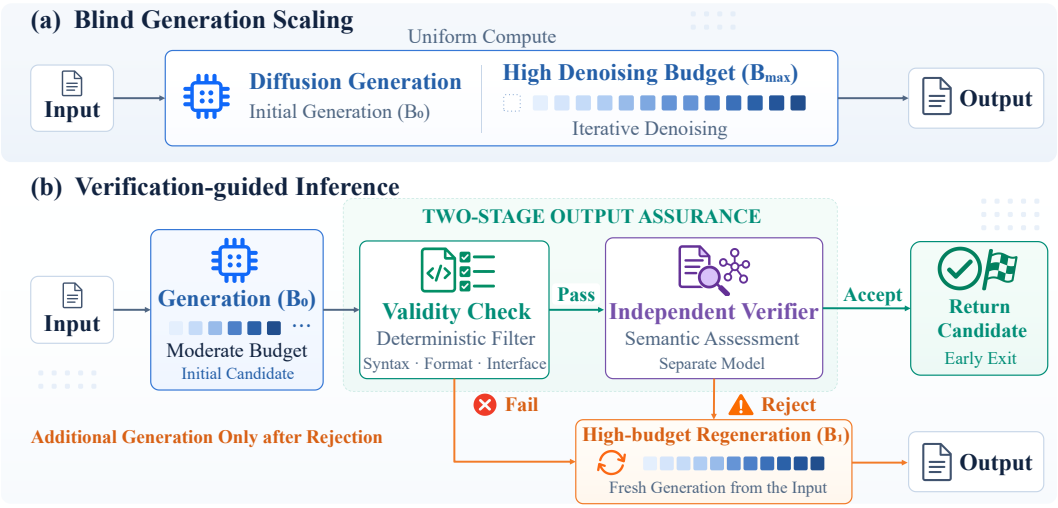


Figure 2: **Blind generation scaling and verification-guided inference.** Conventional scaling applies the higher generation budget uniformly to every input. Verification-guided inference first generates a moderate-budget candidate and applies a deterministic validity check. Candidates that fail are escalated immediately, whereas validity-passing candidates are evaluated by an independent verifier. Only candidates that pass both assurance stages terminate at the moderate budget.

3 METHOD

The loud-to-silent-to-correct transition shows that model outputs occupy different reliability regimes as generation compute increases. At a moderate budget, some candidates remain overtly invalid, some are surface-valid but semantically unreliable, and some are already reliable enough to return. This motivates us to decouple test-time inference into two roles: *candidate generation* and *output assurance*. We call this procedure *verification-guided inference*. The model first generates a moderate-budget candidate, applies a lightweight deterministic validity check, and sends only validity-passing candidates to an independent semantic verifier. Candidates that pass both stages terminate immediately; all others receive fixed higher-budget regeneration. Unlike input-side adaptive allocation, the additional-compute decision is made only after observing the generated candidate.

Moderate-Budget Generation. Given an input x, a diffusion language model G_θ generates

$$y_0 = G_\theta(x; B_0), \tag{1}$$

where B_0 is the initial denoising budget. We evaluate two fixed moderate-budget operating points, $B_0 \in \{64, 128\}$, and use $B_1 = 256$ for higher-budget generation. The initial stage exposes the candidate state before every input is committed to the most expensive generation configuration.

Two-Stage Output Assurance. We first apply a deterministic validity check

$$v(x, y_0) \in \{0, 1\}, \tag{2}$$

where $v = 1$ indicates that the candidate satisfies directly observable validity requirements. For code generation, these include parsing, loading by the evaluation harness, and exposing the required program interface. This stage does not use benchmark correctness tests. Candidates with $v = 0$ are escalated immediately without invoking the semantic verifier. In the standalone validity ablation at $B_0 = 64$, this stage reduces independent-verifier calls from 100% to 54.5% while preventing overtly invalid moderate-budget candidates from being returned without escalation; we provide the corresponding results in Appendix F. Candidates with $v = 1$ are evaluated by an independent verifier V_ϕ. In our experiments, the verifier directly produces a binary acceptance decision

$$a(x, y_0) \in \{0, 1\}, \tag{3}$$

where $a = 1$ indicates that the candidate is judged reliable enough to return. The verifier is deliberately separate from the diffusion generator: generation constructs a candidate, whereas verification

evaluates whether the candidate should be trusted. The loud–silent taxonomy is used to characterize the scaling phenomenon, not as an inference-time classifier. The validity stage handles directly observable failures, while the independent verifier resolves the remaining uncertainty among surface-valid candidates. Verification therefore determines whether the current candidate is reliable enough to stop, rather than predicting whether additional generation will necessarily repair it. Its role is to avoid spending higher-budget generation uniformly on candidates that are already judged reliable.

Selective Escalation. Candidates that fail either assurance stage are independently regenerated with the fixed higher budget:

$$y_1 = G_\theta(x; B_1). \tag{4}$$

The final output is

$$\hat{y} = \begin{cases} y_0, & v(x, y_0) = 1 \text{ and } a(x, y_0) = 1, \\ y_1, & \text{otherwise.} \end{cases} \tag{5}$$

Both B_0 and B_1 are fixed across inputs; only the decision to invoke B_1 is adaptive. Verification-guided inference therefore conditions additional generation compute on the observed candidate rather than assigning a task-specific budget before decoding.

Compute Utilization. Let $C_G(B_0)$ denote the initial generation cost, C_{val} the deterministic validity-check cost, C_V the cost of one verifier call, and $C_E = C_G(B_1)$ the higher-budget regeneration cost. Let p_{valid} be the fraction of candidates that invoke the verifier and p_{esc} the fraction escalated by either assurance stage. The expected inference cost is

$$\mathbb{E}[C_{\text{VG}}] = C_G(B_0) + C_{\text{val}} + p_{\text{valid}} C_V + p_{\text{esc}} C_E. \tag{6}$$

Blind generation scaling instead incurs

$$C_{\text{blind}} = C_G(B_{\max}). \tag{7}$$

Verification-guided inference is computationally favorable when

$$C_{\text{val}} + p_{\text{valid}} C_V + p_{\text{esc}} C_E < C_G(B_{\max}) - C_G(B_0). \tag{8}$$

We evaluate this trade-off end to end using correctness, silent-failure rate, and average inference compute, including all generation, validity-check, verification, and regeneration costs.

4 Experimental Setup

Models, tasks, and inference budgets. We use LLaDA-8B-Instruct (Nie et al., 2025) as the primary diffusion language model and vary its denoising budget over $B \in \{8, 16, 32, 64, 128, 256\}$, where each step corresponds to one full-sequence model evaluation. Dream-v0-Instruct-7B (Ye et al., 2025) provides an independent diffusion-family replication, while Dream-Coder-v0-Instruct-7B (Xie et al., 2025) is used for additional mathematical and security-sensitive evaluations. For autoregressive controls, we evaluate Qwen2.5-7B-Instruct (Yang et al., 2024) under self-refinement (Madaan et al., 2023) and best-of-N scaling (Brown et al., 2024), together with DeepSeek-Coder-6.7B-Instruct (Guo et al., 2024) as a second model family. Our primary benchmark combines HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021b) for 664 Python code-generation tasks. We further use 300 GSM8K (Cobbe et al., 2021) problems, CWEval (Peng et al., 2025) for joint functionality and security evaluation, and EvalPlus (Liu et al., 2023) for stronger re-evaluation of stored code generations. Detailed model configurations, decoding settings, task construction, and prompts are provided in Appendix A.

Failure and verification protocol. We classify outputs as *correct*, *loud failures*, or *silent failures*. For code generation, loud failures are malformed, non-executable, or otherwise directly invalid outputs, whereas silent failures are structurally valid and executable but semantically incorrect. On GSM8K, the distinction is determined by whether a well-formed final numeric answer can be extracted. CWEval additionally identifies *silent-vulnerable* outputs that pass the functionality oracle but fail the security oracle. For verifier analysis, we evaluate stored LLaDA candidates at $B \in \{64, 128, 256\}$. Self-verification uses LLaDA itself, while independent verification uses a single binary accept/reject judgment from Qwen2.5-7B-Instruct. End-to-end verification-guided inference follows the two-stage assurance procedure in Section 3: candidates first undergo

Table 1: **Failure composition under increasing diffusion generation compute.** Results are reported on HumanEval and MBPP. All entries are percentages; Silent Share denotes the fraction of incorrect outputs that are silent.

	LLaDA-8B				Dream-7B			
B	Correct	Loud	Silent	S. Share	Correct	Loud	Silent	S. Share
8	0.9	**93.2**	5.9	5.9	5.6	**73.5**	20.9	22.2
64	22.7	45.5	31.8	41.1	25.3	42.0	32.7	43.8
256	**41.1**	**13.1**	**45.8**	**77.7**	**54.5**	**0.5**	**45.0**	**99.0**

the task-specific deterministic validity check, and only validity-passing candidates invoke the independent verifier. We evaluate $B_0 \in \{64, 128\}$ and independently regenerate rejected candidates at $B_1 = 256$. The GSM8K failure-transition results in Figure 1(c) use greedy decoding ($T = 0$). The end-to-end GSM8K evaluation uses a separate $T = 0.2$ generation run over the same 300 tasks, with full responses retained for verification. Exact failure definitions, validity checks, execution or answer-extraction oracles, and verification prompts are provided in Appendix B.

Evaluation and compute accounting. We report correctness, absolute loud- and silent-failure rates, and *Silent Share*, the fraction of incorrect outputs that are silent. For CWEval, we additionally report functionality-passing but vulnerable outputs. Verifier performance is measured by loud-failure rejection, silent-failure detection, and false rejection of correct candidates. Diffusion generation cost is measured in full-sequence model evaluations, while verifier overhead is reported in token-forwards to provide a common inference-cost proxy across the 8B generator and 7B verifier. End-to-end cost includes initial generation, deterministic validity checking, selective verifier calls, and all triggered high-budget regenerations, following Eq. equation 6. We jointly report correctness, silent-failure rate, and average inference compute. Further compute and statistical details are in Appendix C.

5 Experiments

Our experiments first ask whether increasing diffusion generation compute changes only the overall error rate or also the distribution of model outcomes. We then examine whether the resulting silent-failure regime is a persistent endpoint or an intermediate stage of scaling, test the robustness and operational significance of this transition, and finally evaluate whether the observed failure structure can be exploited to improve inference.

Increasing generation compute shifts the dominant failure regime from loud to silent. On the combined HumanEval and MBPP benchmark, increasing the LLaDA denoising budget from 8 to 256 steps raises correctness from 0.9% to 41.1%. Accuracy alone, however, hides a substantially larger redistribution in the remaining outcomes. Loud failures fall from 93.2% to 13.1%, whereas silent failures increase from 5.9% to 45.8%. Among incorrect outputs, the fraction that is silent consequently rises from 5.9% to 77.7%. The scaling trajectory therefore shifts from a regime dominated by overtly invalid failures toward one in which most remaining errors are surface-valid but semantically incorrect. The same pattern appears with Dream-v0-Instruct-7B. Its correctness rises from 5.6% at 8 steps to 54.5% at 256 steps, while loud failures fall from 73.5% to 0.5%. Silent failures simultaneously increase from 20.9% to 45.0%, leaving 99.0% of the remaining failures silent at the largest budget. These code-generation experiments therefore reveal the loud-to-silent portion of the scaling trajectory, but they do not yet show whether the silent regime is the endpoint of scaling or an intermediate stage before correctness becomes dominant.

Silent failures form an intermediate regime before correctness becomes dominant. GSM8K exposes a later part of the same scaling trajectory. As shown in Figure 1(c), LLaDA correctness increases from 10.7% at $B = 8$ to 80.0% at $B = 256$, while loud failures fall from 66.0% to 7.0%. The absolute silent-failure rate initially rises from 23.3% to 38.3% at $B = 16$, but then declines to 13.0% at $B = 256$. The conditional silent share similarly rises from 26.1% at $B = 8$, reaches 72.3% at $B = 64$, and falls to 65.0% at the largest budget. This trajectory completes the failure-distribution shift suggested by the code-generation results. At low compute, outcomes are dominated by loud failures. As generation compute increases, loud failures disappear faster than semantic errors, shifting the remaining failures toward the silent category. At still higher compute,

Table 2: **Generality and controls for failure-composition changes.** Values span the tested compute range; Silent Share is measured among incorrect outputs.

Setting	Compute range	Correct	Silent Share
Diffusion models and evaluation controls			
LLaDA, GSM8K	$8 \rightarrow 256$	$\mathbf{10.7 \rightarrow 80.0}$	$\mathbf{26.1 \rightarrow 65.0}$
Dream-Coder, GSM8K	$8 \rightarrow 128$	$\mathbf{2.7 \rightarrow 54.3}$	$\mathbf{15.4 \rightarrow 17.5}$
LLaDA, EvalPlus	$8 \rightarrow 256$	$\mathbf{1.5 \rightarrow 42.3}$	$\mathbf{6.8 \rightarrow 83.5}$
Dream, EvalPlus	$8 \rightarrow 256$	$\mathbf{5.7 \rightarrow 56.7}$	$\mathbf{24.6 \rightarrow 98.2}$
LLaDA, $T = 0.2$	$8 \rightarrow 256$	$\mathbf{0.8 \rightarrow 40.5}$	$\mathbf{3.2 \rightarrow 75.4}$
Autoregressive controls			
Qwen, best-of-N oracle	$1 \rightarrow 8$	$69.1 \rightarrow 80.9$	$\sim \mathbf{99} \rightarrow \sim \mathbf{99}$
DeepSeek, refinement	$0 \rightarrow 4$	$65.7 \rightarrow 66.4$	$\mathbf{95.2 \rightarrow 96.4}$

correctness increasingly catches up and the absolute silent-failure rate begins to decline. Silent failure is therefore not simply an error type that grows monotonically with compute; it marks an intermediate regime in which surface validity has improved substantially while semantic correctness still lags behind. Dream-Coder-7B does not reach the same stage within the tested GSM8K range. Its correctness improves from 2.7% to 54.3% by $B = 128$, while 37.7% of generations remain loud and the silent share is 17.5%. Table 2 places this setting alongside the main robustness controls. The location of the silent regime therefore depends on the capability of the model–task pair rather than on a universal denoising budget: different model–task pairs may occupy different portions of the same scaling trajectory over the compute range we observe.

The transition survives decoding and evaluation controls. We next test whether the observed loud-to-silent redistribution is specific to a particular decoding or evaluation configuration. Dream and LLaDA use different default decoding configurations, so we rerun the complete LLaDA code benchmark at temperature 0.2. The silent share at $B = 256$ remains 75.4%, close to the 77.7% obtained under greedy decoding, and the same redistribution remains visible across the denoising range in Table 2. The change is therefore not driven by the greedy-versus-stochastic decoding difference between the two diffusion models. Stronger correctness tests produce a similar picture. Re-grading the stored HumanEval and MBPP generations with EvalPlus yields a silent share of 83.5% for LLaDA and 98.2% for Dream at $B = 256$. Because EvalPlus covers a 388-task subset and no new generations are produced, this control tests whether the failure-composition shift persists under stronger correctness evaluation without introducing generation variability. Prompt construction changes the absolute level of the measured failure rates but preserves the qualitative increase in silent failures across denoising budgets. On the same 24 CWEval tasks, both whole-file and anchor-splice prompting exhibit increasing silent share as compute grows, although their absolute levels differ. Full prompt-sensitivity results are reported in Appendix G. The autoregressive controls provide a different reference point. Qwen2.5-7B begins with almost all remaining failures already in the silent category, so increasing best-of-N improves oracle correctness from 69.1% to 80.9% without producing a comparable redistribution between loud and silent errors. DeepSeek-Coder exhibits similarly little movement across self-refinement rounds. These comparisons are not capability-matched tests of diffusion against autoregressive generation, but the pronounced loud-to-silent movement observed along diffusion denoising is not reproduced by the autoregressive compute axes tested here. The remaining question is whether this shift is merely taxonomic or whether silent failures can survive ordinary acceptance criteria in operationally meaningful ways.

Silent failures can satisfy functionality while remaining vulnerable. CWEval provides separate functionality and security oracles, allowing this distinction to be tested directly. We use the matched anchor-splice protocol described in Appendix A so that malformed benchmark scaffolding is not conflated with semantic or security failures. Table 3 reports the higher-budget operating points. For LLaDA, the fraction of all generations that pass functionality while failing the security oracle rises from 11.5% at $B = 64$ to 21.9% at $B = 256$. At the latter budget, 18.2% of generations satisfy both functionality and security. For Dream-Coder, evaluated on the full 72-task pool, 37.5% of generations are correct and secure at $B = 256$, while another 29.2% pass functionality but fail the security oracle. Among its remaining failures, 46.7% are functionality-passing yet vulnerable. Because the two models are evaluated on different CWEval task pools, these rates are not intended as a direct capability comparison. At this stage, an output can therefore appear usable under ordinary functional

Table 3: **Security-sensitive generation on CWEval.** Results use the matched anchor-splice protocol with eight generations per task and budget. Silent-Vulnerable denotes functionality-passing outputs that fail the security oracle. All entries are percentages.

	LLaDA-8B (24 Python)		**Dream-Coder-7B** (72 tasks)	
B	Correct & Secure	Silent-Vulnerable	Correct &Secure	Silent-Vulnerable
64	7.3	11.5	4.2	4.2
128	9.4	19.3	20.8	16.7
256	**18.2**	**21.9**	**37.5**	**29.2**

Table 4: **Self-verification versus independent verification on stored LLaDA generations.** Self-verification uses 64 additional LLaDA denoising steps; independent verification uses one Qwen2.5-7B pass. All entries are percentages.

	Self-Verification		**Independent Verification**		
Gen. B	Silent Detection	False Reject	Loud Reject	Silent Detection	False Reject
64	16.6	2.0	**94.0**	**82.5**	21.9
128	7.4	2.6	**87.5**	**76.2**	23.8
256	4.6	2.2	**77.0**	**72.4**	21.6

evaluation while retaining an error that is only exposed by a stronger assurance criterion. Once such candidates pass the obvious validity checks, the problem shifts from producing a plausible output to determining whether that output can actually be trusted. This distinction also affects verification: on a separate generation set from the same 24-task CWEval subset, a generic correctness verifier detects only 10.5%–21.1% of vulnerable outputs, with wide task-level bootstrap uncertainty (e.g., 10.5%, 95% CI $[0.0, 41.2]$). In contrast, a security-aware verification prompt detects 94.7%–95.5%, with substantially higher recall (e.g., 94.7%, 95% CI $[76.5, 100.0]$; full results in Appendix H).

Independent verification exposes errors that additional self-verification misses. We first give the assurance computation back to the diffusion generator itself. LLaDA candidates generated at $B \in \{64, 128, 256\}$ are passed back to the same model for self-verification, with verification budgets ranging from 8 to 64 additional denoising steps. Even with 64 verification steps, self-verification remains conservative but weak at detecting silent errors: false rejection of correct candidates is only 2.0%–2.6%, while silent-error detection reaches just 16.6%, 7.4%, and 4.6% for candidates generated at $B = 64$, 128, and 256, respectively. To characterize the independent verifier itself separately from the staged end-to-end routing policy, we apply Qwen2.5-7B-Instruct to all stored candidates from the standalone scaling runs, including loud failures, silent failures, and correct outputs. These offline statistics characterize verifier behavior on a fixed candidate set rather than reconstructing the end-to-end routing results reported below. As reported in Table 4, a single verification pass rejects 94.0%, 87.5%, and 77.0% of loud failures at $B = 64$, 128, and 256, respectively. It also rejects 82.5%, 76.2%, and 72.4% of silent failures at the same budgets. The verifier also rejects 21.9%, 23.8%, and 21.6% of correct candidates, so its decisions cannot be treated as a free oracle: false rejection increases the frequency of unnecessary high-budget regeneration. At $B = 256$, one independent verification pass requires approximately 2.3% of the token-forward compute of generation, with the full accounting given in Appendix C. Strong error detection alone, however, does not establish whether staged assurance improves the complete inference process; verifier cost, false rejection, and all triggered regenerations must be evaluated together end to end.

Verification-guided inference improves reliability and efficiency end to end. The preceding experiments show that moderate-budget generation already produces candidates with different reliability states: some remain overtly invalid, some are surface-valid but semantically incorrect, and others are already correct. We therefore evaluate whether conditioning additional generation compute on the observed candidate improves the complete inference process. For code generation, overtly invalid candidates are first rejected by the deterministic validity check, while validity-passing candidates are evaluated by the independent verifier. We consider two verification-guided operating points, $B_0 = 64$ and $B_0 = 128$, representing lower- and higher-compute settings, respectively. Each routing configuration is evaluated with a separate end-to-end generation run rather than by re-

Table 5: **End-to-end comparison on HumanEval and MBPP.** All methods return an answer for every task. Compute is reported in generation-equivalent units; Saving is relative to blind $B = 256$.

Method	Compute	Saving	Correct	Silent
Blind $B = 256$	256	–	41.1	45.8
Exec. cascade	190	26.0%	40.5	–
VG $B_0 = 64 \rightarrow 256$	**207**	**19.1%**	**44.0**	**43.1**
VG $B_0 = 128 \rightarrow 256$	**243**	**5.1%**	**48.0**	**34.0**

playing the stored candidates used in the standalone scaling and verifier analyses, so the policy-level correctness, failure rates, and compute are measured directly from these runs. As shown in Table 5, the $B_0 = 64$ configuration reaches 44.0% correctness with a 43.1% silent-failure rate at an average compute of 207, compared with 41.1% correctness, 45.8% silent failures, and compute 256 under blind $B = 256$ generation. The higher-compute $B_0 = 128$ configuration further improves correctness to 48.0% and reduces the silent-failure rate to 34.0%, while requiring an average compute of 243. To isolate the contribution of semantic verification beyond deterministic validity filtering, we also evaluate an executability-only cascade at $B_0 = 128$, where executable candidates terminate immediately and only execution failures are regenerated at $B = 256$. This baseline reduces average compute by 26% but achieves only 40.5% correctness ($269/664$), compared with 41.1% ($273/664$) under blind $B = 256$, indicating that validity-based early termination alone does not recover the reliability gains obtained when semantic verification is added. The same end-to-end advantage also extends beyond code generation. In the separate $T = 0.2$ GSM8K run, blind $B = 256$ generation achieves 80.3% correctness at an average generation compute of 256, whereas verification-guided inference reaches 84.3% correctness at 170.8 average compute for $B_0 = 64$ and 84.0% correctness at 201.7 average compute for $B_0 = 128$ (see Appendix D for full results and latency measurements).

Wall-clock latency. Because verification introduces additional inference stages, we separately measure wall-clock cost on a single H100 with both neural models resident in memory, CUDA synchronization, and warmup excluded; full results are reported in Appendix E. LLaDA-8B takes 5.849 s on average for $B = 256$ generation, whereas one Qwen2.5-7B verification pass takes only 39.2 ms, or 0.67% of the generation latency. As a complementary reference, the executability-only cascade at $B_0 = 128$ terminates 75.9% of requests after the first stage and reduces mean policy latency from 5.849 s to 4.336 s while retaining 269 of the 273 correct solutions from blind $B = 256$. These measurements distinguish verifier overhead from the latency savings enabled by selective regeneration.

6 Conclusion and Limitations

We study how model outcomes evolve as test-time generation compute increases and identify a compute-dependent transition in diffusion language models: failure composition shifts from loud toward silent failures before correctness becomes dominant. This reveals that *surface validity can improve before semantic correctness*, creating an intermediate regime in which outputs appear usable while remaining semantically unreliable; on security-sensitive tasks, such outputs can even pass functionality tests while retaining vulnerabilities. Motivated by this transition, we introduce *verification-guided inference*, which separates candidate generation from output assurance and conditions higher-budget generation on the observed candidate rather than applying additional compute uniformly. Independent verification detects 72–83% of silent failures with a low-cost pass, while end-to-end verification-guided inference improves correctness from 41.1% to 48.0%, reduces silent failures from 45.8% to 34.0%, and lowers average generation-equivalent compute from 256 to 243. Our evidence is strongest for diffusion language models on code generation, mathematical reasoning, and security-sensitive programming, and the location of the silent-failure regime varies across model–task pairs. The autoregressive controls do not establish that this transition is unique to diffusion generation, and the loud–silent distinction depends on the available evaluation oracle. Our current inference policy also uses fixed B_0 and B_1, task-specific validity checks where available, and a single independent verifier; false rejection remains possible, and rejection does not guarantee successful higher-budget regeneration. Finally, token-forward compute is a hardware-independent proxy that may not fully capture realized deployment costs under batching, distributed execution, or service-based inference settings.

AI USE STATEMENT

In preparing this submission, we used large language models (LLMs) as general writing assistants. Specifically, LLMs were used to improve the clarity and readability of the manuscript, including sentence restructuring, grammatical editing, and concision. All research ideas, method design, experiments, analyses, and final scientific decisions were independently developed, conducted, and verified by the authors. We reviewed all AI-assisted text and take responsibility for the final content of this work.

ETHICS STATEMENT

This work does not involve human subjects or private user data. All experiments use existing research benchmarks and language models, and code-generation outputs are evaluated in benchmark-controlled environments. The security-sensitive experiments on CWEval are intended to study reliability failures and defensive verification, rather than to facilitate exploitation of real-world systems. Third-party datasets, models, and tools remain subject to their respective licenses and usage conditions.

REPRODUCIBILITY STATEMENT

The main text specifies the proposed inference procedure, evaluation protocol, and reported metrics. The appendix documents model and task configurations, prompting and decoding settings, failure definitions, verification procedures, compute accounting, statistical analysis, latency measurement, and additional ablation experiments. Together, these details specify the experimental conditions and evaluation procedures used to obtain the reported results.

REFERENCES

Jacob Austin, Daniel D Johnson, Jonathan Ho, Daniel Tarlow, and Rianne Van Den Berg. Structured denoising diffusion models in discrete state-spaces. *Advances in neural information processing systems*, 34:17981–17993, 2021a.

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*, 2021b.

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*, 2024.

Huiwen Chang, Han Zhang, Lu Jiang, Ce Liu, and William T Freeman. Maskgit: Masked generative image transformer. In *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 11305–11315. IEEE, 2022.

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.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Yaoxuan Dou and Yang Shu. Faster but different: Diagnosing and controlling content drift in accelerated multimodal diffusion language models. *arXiv preprint arXiv:2607.29079*, 2026.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. DeepSeek-Coder: When the large language model meets programming – the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. In *International conference on learning representations*, volume 2024, pp. 32808–32824, 2024.

Fan Li, Yu Gu, Zhigang Wang, Fangling Leng, Zhenghao Liu, and Ge Yu. Towards efficient and effective diffusion language model inference via semantic-aware adaptive denoising. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 17988–18002, 2026.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *International Conference on Learning Representations*, volume 2024, pp. 39578–39601, 2024.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in neural information processing systems*, 36:21558–21572, 2023.

Aaron Lou, Chenlin Meng, and Stefano Ermon. Discrete diffusion modeling by estimating the ratios of the data distribution. *arXiv preprint arXiv:2310.16834*, 2023.

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.

Shen Nie, Fengqi Zhu, Zebin You, Xiaolu Zhang, Jingyang Ou, Jun Hu, Jun Zhou, Yankai Lin, Ji-Rong Wen, and Chongxuan Li. Large language diffusion models. *arXiv preprint arXiv:2502.09992*, 2025.

Jinjun Peng, Leyi Cui, Kele Huang, Junfeng Yang, and Baishakhi Ray. Cweval: Outcome-driven evaluation on functionality and security of llm code generation. In *2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*, pp. 33–40. IEEE, 2025.

Michael Rottoli, Subhankar Roy, and Stefano Paraboschi. Predict-then-diffuse: Adaptive response length for compute-budgeted inference in diffusion llms. *arXiv preprint arXiv:2605.04215*, 2026.

Subham S Sahoo, Marianne Arriola, Yair Schiff, Aaron Gokaslan, Edgar Marroquin, Justin T Chiu, Alexander Rush, and Volodymyr Kuleshov. Simple and effective masked diffusion language models. *Advances in Neural Information Processing Systems*, 37:130136–130184, 2024.

Ivan Sedykh, Nikita Sorokin, and Valentin Malykh. Not all denoising steps are equal: Model scheduling for faster masked diffusion language models. *arXiv preprint arXiv:2604.02340*, 2026.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

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.

Zhihui Xie, Jiacheng Ye, Lin Zheng, Jiahui Gao, Jingwei Dong, Zirui Wu, Xueliang Zhao, Shansan Gong, and Xin Jiang. Dream-Coder 7b: An open diffusion language model for code. *arXiv preprint arXiv:2509.01142*, 2025.

An Yang, Beichen Zhang, Binyuan Hui, Bofei Gao, Bowen Yu, Chengpeng Li, Dayiheng Liu, Jianhong Tu, Jingren Zhou, Junyang Lin, et al. Qwen2. 5-math technical report: Toward mathematical expert model via self-improvement. *arXiv preprint arXiv:2409.12122*, 2024.

Jiacheng Ye, Zhihui Xie, Lin Zheng, Jiahui Gao, Zirui Wu, Xin Jiang, Zhenguo Li, and Lingpeng Kong. Dream 7b: Diffusion large language models, 2025. *URL https://arxiv. org/abs/2508.15487*, 2025.

APPENDIX CONTENTS

A Model, Task, and Prompt Configurations 13

B Failure Taxonomy and Verification Protocol 14

C Compute Accounting and Statistical Details 15

D End-to-End Generalization on GSM8K 16

E Wall-Clock Latency Analysis 17

F Effect of the Deterministic Validity Stage 19

G Prompt-Sensitivity Control 19

H Security-Aware Verification on CWEval 20

A Model, Task, and Prompt Configurations

Diffusion models. We use LLaDA-8B-Instruct (Nie et al., 2025) as the primary masked diffusion language model. Unless otherwise specified, we evaluate six denoising budgets, $B \in \{8, 16, 32, 64, 128, 256\}$, with a generation length of 256 tokens and a block length of 32. The primary LLaDA experiments use greedy decoding and low-confidence remasking. Dream-v0-Instruct-7B (Ye et al., 2025) provides an independent diffusion-family replication and is evaluated over the same compute grid using MaskGIT+ decoding (Chang et al., 2022) with temperature 0.2, top-$p = 0.95$, and algorithmic temperature 0. Dream performs poorly under greedy decoding in our preliminary accuracy-based probe, so we use its recommended stochastic configuration and separately rerun LLaDA at temperature 0.2 over all 664 code-generation tasks and all six budgets to control for this decoding asymmetry. Dream-Coder-v0-Instruct-7B is used for the additional mathematical and security-sensitive evaluations.

Autoregressive controls. Qwen2.5-7B-Instruct (Yang et al., 2024) is evaluated along two test-time compute axes. Self-refinement uses $k \in \{0, 1, 2, 4\}$ refinement rounds, while best-of-N uses $N \in \{1, 2, 4, 8\}$ sampled candidates with temperature 0.8. DeepSeek-Coder-6.7B-Instruct (Guo et al., 2024) provides a second autoregressive model family under the same self-refinement schedule. These experiments are designed to test whether the observed change in failure composition follows automatically from increasing test-time computation; they are not intended as capability-matched comparisons with the diffusion models.

Code-generation tasks. Our primary benchmark combines all 164 HumanEval (Chen et al., 2021) problems with all 500 MBPP (Austin et al., 2021b) problems, yielding 664 Python code-generation tasks. HumanEval candidates are evaluated using the benchmark unit tests in a sandboxed subprocess with a 10-second timeout. MBPP uses the benchmark assertions under the same execution harness. Because the natural-language MBPP descriptions do not always specify the required function name, the corresponding assertions are included in the model prompt.

Mathematical reasoning and stronger tests. For mathematical reasoning, we use the first 300 examples from the GSM8K (Cobbe et al., 2021) test set. Correctness is determined by exact match on the extracted final numeric answer. To test whether the code-generation results arise from weak unit tests, we additionally re-grade the same stored generations using the strengthened EvalPlus (Liu et al., 2023) suites on the 388 HumanEval and MBPP tasks for which EvalPlus coverage is available. No new generations are produced for this control.

Security-sensitive tasks. We use CWEval (Peng et al., 2025) to evaluate failures under separate functionality and security criteria. A task is admitted only when its secure reference implementation passes both oracles. For JavaScript, C, and Go, we additionally require the supplied insecure implementation to pass functionality while failing the security oracle, ensuring that the security oracle can distinguish vulnerable behavior. We additionally exclude `cwe_1333_0` because its timing-dependent ReDoS oracle fails on the benchmark's own secure reference implementation. After filtering, 72 tasks remain: 24 Python, 13 JavaScript, 20 C, and 15 Go. LLaDA is evaluated on the 24-task Python subset, while Dream-Coder is evaluated on the full 72-task filtered pool. For stochastic CWEval evaluation, we generate eight samples for each task and denoising budget.

Prompting protocols. Diffusion and autoregressive models receive byte-identical prompts in the primary HumanEval and MBPP comparison. HumanEval prompts ask the model to complete the target Python function and return only the function inside a Python code block. MBPP prompts contain the natural-language task together with the benchmark assertions. GSM8K prompts request brief reasoning followed by a final numeric answer in the standard benchmark answer format.

For the main CWEval evaluations, we use an anchor-splice protocol rather than requesting complete-file regeneration. The source prefix is preserved through the benchmark's solution marker, the model generates only the solution body, and the original entry-point suffix is restored before compilation and evaluation. This prevents truncation or accidental removal of benchmark infrastructure from being counted as semantic model failures. We separately evaluate whole-file versus anchor-splice prompting as a prompt-sensitivity control.

B Failure Taxonomy and Verification Protocol

Code-generation failure taxonomy. For HumanEval and MBPP, we classify each generated program as correct, a loud failure, or a silent failure. A program is correct if it passes all benchmark tests. Loud failures are outputs that fail the pre-semantic validity checks, including malformed code, missing executable output, absent required entry points, or other structural failures that prevent the candidate from entering functional evaluation. Silent failures are candidates that pass these validity checks but fail the benchmark correctness oracle. This category therefore includes semantically incorrect programs as well as runtime failures that arise only when the candidate is exercised on benchmark test inputs.

Mathematical failure taxonomy. For GSM8K, we apply the same distinction without relying on program syntax. A response is classified as correct if the extracted final numeric answer matches the reference answer. A response is loud when no well-formed final answer can be extracted from the generated text. If a valid numeric answer can be extracted but is incorrect, the response is classified as a silent failure. This definition allows us to test the same validity-versus-correctness distinction outside code generation.

Security-aware failure taxonomy. CWEval provides separate functionality and security oracles, allowing us to distinguish ordinary correctness failures from security failures. We consider an output correct and secure when it passes both oracles. Outputs that cannot be compiled, executed, or exposed through the required interface are treated as loud failures. Executable outputs that fail the functionality oracle are classified as ordinary silent failures. We additionally define a *silent-vulnerable* output as one that passes all functionality tests but fails the security oracle. Such outputs are particularly important because they would be accepted under functionality-only evaluation despite retaining a security vulnerability.

Self-verification. We evaluate whether the diffusion generator can detect its own silent failures using LLaDA candidates produced at $B \in \{64, 128, 256\}$. The generated candidate and original task are provided back to LLaDA, which is asked to determine whether the candidate is correct. We vary the self-verification budget over $8, 16, 32$, and 64 additional denoising steps. The verifier is instructed to produce a binary yes-or-no judgment, and responses that do not conform to the expected decision format are recorded separately rather than automatically treated as acceptance or rejection. At the largest self-verification budget of 64 steps, silent-error detection is 16.6%, 7.4%, and 4.6% for candidates generated at $B = 64$, 128, and 256, respectively. False rejection of correct candidates remains much lower, at 2.0%, 2.6%, and 2.2%. Thus, self-verification operates conservatively, but its low false-rejection rate is accompanied by weak detection of silent failures.

Independent verification. We use Qwen2.5-7B-Instruct as the independent verifier for the same stored LLaDA candidates. For the standalone verifier analysis, all stored candidates are passed to Qwen2.5-7B-Instruct regardless of their failure category. The verifier receives both the original task and the generated candidate and returns a binary judgment indicating whether the candidate should be accepted. The verifier directly produces a binary accept/reject decision, so no task-specific threshold is fitted. We characterize the verifier using three quantities: the fraction of loud failures that it rejects, the fraction of silent failures that it rejects, and the fraction of correct outputs that it incorrectly rejects. The first measures whether the verifier recognizes overtly invalid outputs when they are presented to it directly, the second measures detection of valid but semantically incorrect candidates, and the third captures false rejection of correct outputs. This standalone analysis is separate from the end-to-end routing policy, where candidates that fail the deterministic validity check are rejected before the independent verifier is invoked.

Verification-guided escalation. The end-to-end code-generation procedure uses a staged assurance policy with global generation budgets. We evaluate $B_0 \in \{64, 128\}$ as moderate-budget initial generation points and use $B_1 = 256$ as the higher-budget regeneration configuration. Every input first receives the same B_0. The resulting candidate then undergoes the deterministic validity check used by the execution harness. Candidates that fail this check are rejected immediately without invoking the independent verifier. Candidates that pass are evaluated by Qwen2.5-7B-Instruct. Only candidates that pass both the validity stage and independent verification terminate at B_0; all others are independently regenerated at B_1. Thus, the pipeline does not predict loud or silent failure labels at inference time. It first exploits failure signals that are directly observable through deterministic checks and then asks whether the remaining surface-valid candidate is reliable enough to return.

C Compute Accounting and Statistical Details

Generation compute. For diffusion language models, we use the number of denoising steps as the primary test-time generation-compute axis. Each denoising step requires a full-sequence model evaluation, so increasing B directly increases the amount of generation computation. Unless otherwise specified, diffusion experiments use the common budget grid $B \in \{8, 16, 32, 64, 128, 256\}$. This controlled compute axis allows us to study how output correctness and failure composition change as generation computation increases.

Assurance cost. The code-generation pipeline contains two assurance stages. The deterministic validity check uses the execution harness to identify candidates that are already overtly unusable and does not invoke a language model. Candidates that pass this stage are evaluated by one Qwen2.5-7B-Instruct verification pass over the original task and generated candidate. We estimate neural verification overhead using token-forward compute rather than exact FLOPs, providing a common proxy for comparing the 8B diffusion generator and the 7B verifier without relying on architecture-specific hardware assumptions. At $B = 256$, one independent verification pass corresponds to approximately 2.3% of the token-forward compute required by generation. Because the end-to-end pipeline invokes the verifier only for candidates that pass the deterministic validity stage, this per-call ratio is distinct from the average verifier cost per task.

End-to-end inference cost. For verification-guided inference, every code-generation example incurs the initial generation cost $C_G(B_0)$ and the deterministic validity-check cost C_{val}. Only candidates that pass the validity check incur the independent verification cost C_V. Candidates rejected either by the validity stage or by the independent verifier additionally incur the higher-budget regeneration cost C_E. Mean inference cost is therefore computed according to Eq. equation 6:

$$\mathbb{E}[C_{\text{VG}}] = C_G(B_0) + C_{\text{val}} + p_{\text{valid}} C_V + p_{\text{esc}} C_E.$$

Here, p_{valid} is the fraction of candidates that reach the independent verifier, while p_{esc} is the fraction that ultimately invoke high-budget regeneration. Correct candidates that are falsely rejected by the verifier are charged for the resulting regeneration in exactly the same way as genuinely incorrect candidates. For the main verification-guided configurations, we evaluate $B_0 \in \{64, 128\}$ and use $B_1 = 256$ for regeneration.

Matched-compute comparison. For each verification-guided configuration, we calculate the average total compute consumed across the complete task set, including moderate-budget generation, deterministic validity checking, all invoked verifier passes, and all triggered high-budget regenerations. Blind generation instead applies its selected generation budget uniformly to every example. At each operating point, we jointly report correctness, silent-failure rate, and mean inference cost. This comparison measures whether avoiding unnecessary verifier calls and high-budget generation on candidates that can already be accepted yields a more favorable reliability–compute trade-off than uniform high-budget generation.

Failure metrics. Let N denote the total number of evaluated examples, and let N_{loud}, N_{silent}, and N_{correct} denote the corresponding outcome counts. We report the absolute loud- and silent-failure rates over all examples. In addition, we report the conditional silent share

$$\frac{N_{\text{silent}}}{N_{\text{loud}} + N_{\text{silent}}},$$

which measures what fraction of the remaining errors are silent. We always report this quantity together with the absolute silent-failure rate because the conditional share can increase simply as loud failures disappear, even when the absolute number of silent failures does not increase.

For independent verification, loud-error rejection is measured as the fraction of loud failures rejected by the verifier when all stored candidates are presented to it offline. Silent-error detection is measured as the fraction of silent failures rejected by the verifier, and false rejection is measured over correct candidates. These verifier-level quantities are distinct from the end-to-end routing policy, where overtly invalid code candidates are removed by the deterministic validity stage before the independent verifier is invoked. For CWEval, we additionally report the fraction of all generations that pass the functionality oracle while failing the security oracle.

Statistical analysis. Confidence intervals are computed by resampling at the task level so that multiple generations associated with the same underlying problem are not treated as independent observations. We use 10,000 bootstrap replicates. For ordinary rates, we use task-level cluster bootstrap

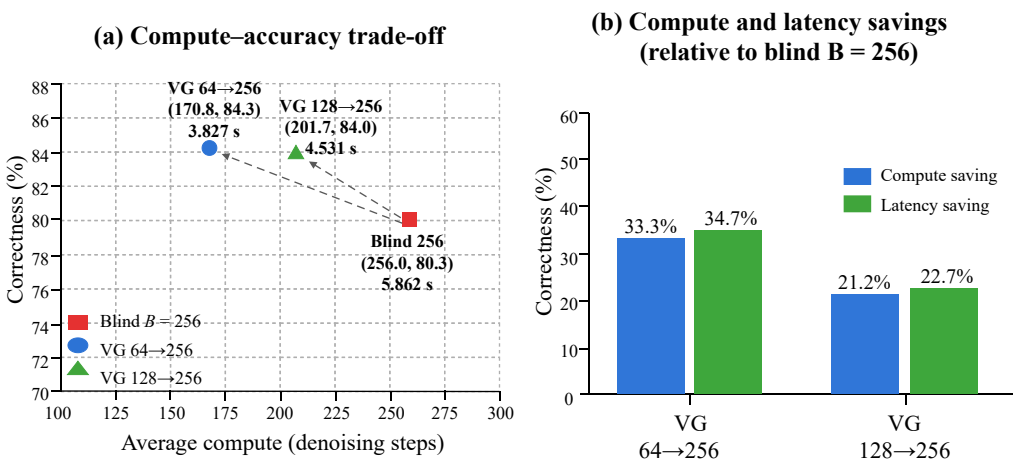


Figure 3: **End-to-end reliability–efficiency trade-off on GSM8K.** (**a**) Correctness versus average generation-equivalent compute for blind $B = 256$ generation and the two verification-guided operating points. Both verification-guided configurations move toward higher correctness with lower average compute. (**b**) Relative compute and wall-clock latency savings with respect to blind $B = 256$ generation.

intervals; ratio-valued quantities such as conditional silent share use bias-corrected and accelerated bootstrap intervals. For stochastic experiments with multiple generations per task, including the CWEval prompt and security-verification controls, the task remains the bootstrap unit. All generations belonging to the same task are resampled together. Thus, for the 24-task CWEval analyses with eight generations per task, inference is based on 24 independent task clusters rather than 192 independent samples.

Reproducibility. Generation seeds are recorded for stochastic configurations, and all task–budget combinations are evaluated under fixed prompting and decoding settings within each comparison. Truncation and execution failures are recorded at the generation level. Hardware and software configurations are kept fixed within each experimental family so that changes across inference budgets reflect test-time computation rather than changes in the execution environment.

D End-to-End Generalization on GSM8K

We further evaluate whether verification-guided inference transfers beyond code generation. We use the same LLaDA-8B generator and Qwen2.5-7B-Instruct independent verifier on the 300-example GSM8K subset described in Appendix A. The same staged assurance procedure used in the main code-generation evaluation is retained. Each moderate-budget candidate first undergoes the deterministic validity check; candidates that fail this stage are rejected immediately, while candidates that pass are evaluated by the independent verifier. Blind generation uses $B = 256$, whereas the verification-guided configurations use $B_0 \in \{64, 128\}$, with all candidates rejected by either assurance stage independently regenerated using $B_1 = 256$.

Figure 3 visualizes the resulting reliability–efficiency trade-off, while Table 6 reports the corresponding numerical results. Blind $B = 256$ generation achieves 80.3% correctness at an average compute of 256 and a mean latency of 5.862 s. Verification-guided inference with $B_0 = 64$ improves correctness to 84.3% while reducing average compute to 170.8 and mean latency to 3.827 s, corresponding to savings of 33.3% and 34.7%, respectively. The $B_0 = 128$ configuration reaches 84.0% correctness at an average compute of 201.7 and mean latency of 4.531 s, reducing compute by 21.2% and latency by 22.7%. The two verification-guided operating points in Figure 3 and Table 6 show that the end-to-end advantage of selective verification is not specific to executable code. In this mathematical reasoning setting, intermediate candidates can already be reliable enough to terminate

Table 6: End-to-end generalization of verification-guided inference on GSM8K. Compute is reported in generation-equivalent denoising steps.

Method	Compute	Correct (%)	Mean Latency (s)
Blind generation			
Blind $B = 256$	256.0	80.3	5.862
Verification-guided inference			
VG $B_0 = 64 \rightarrow 256$	170.8	**84.3**	**3.827**
VG $B_0 = 128 \rightarrow 256$	201.7	84.0	4.531

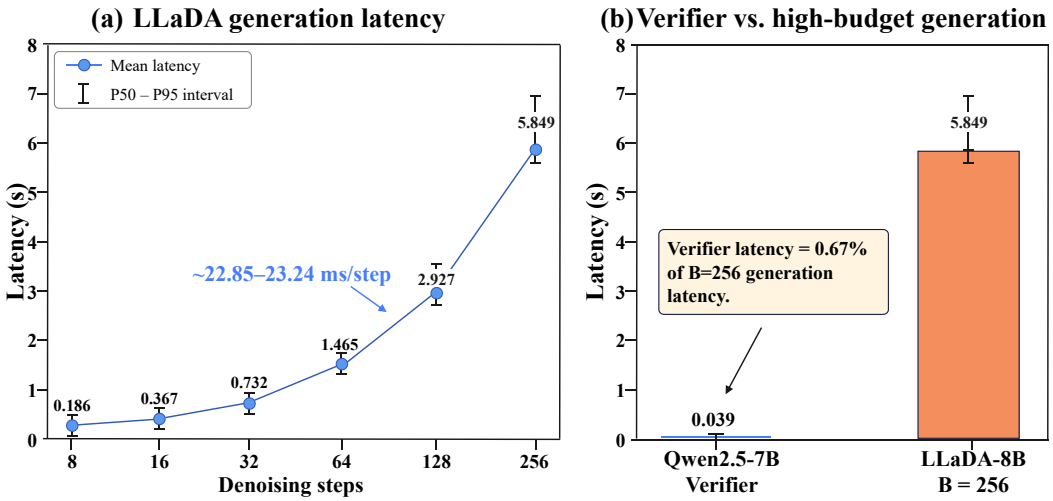


Figure 4: **Stage-level wall-clock latency on one H100.** (**a**) LLaDA-8B generation latency across denoising budgets, with P50–P95 intervals. Generation latency scales nearly linearly with the number of denoising steps. (**b**) Comparison between one Qwen2.5-7B verification pass and $B = 256$ LLaDA-8B generation. The verifier requires only 0.67% of the wall-clock latency of high-budget generation.

after verification, allowing the method to improve correctness while avoiding uniform high-budget generation.

E Wall-Clock Latency Analysis

The main experiments use token-forward computation as the hardware-independent cost proxy, but the staged verification-guided pipeline introduces additional assurance operations whose effect on wall-clock latency may differ from their abstract compute cost. We therefore complement the compute analysis with measurements on a single NVIDIA H100 GPU. Both LLaDA-8B and Qwen2.5-7B remain resident in memory throughout the benchmark, CUDA synchronization is applied before timing, and warmup runs are excluded. Each neural inference configuration contains 38 timed runs. The measurements below separately characterize diffusion generation, independent verification, and the latency consequences of selective high-budget regeneration.

Stage-level latency. Figure 4 visualizes the measured stage-level latency, while Table 7 reports the corresponding numerical results. LLaDA-8B generation latency increases almost linearly with the denoising budget, remaining between 22.85 and 23.24 ms per step. At $B = 256$, generation requires 5.849 s on average, whereas one Qwen2.5-7B verification pass requires only 39.2 ms, or 0.67% of the generation latency. Figure 4 highlights the large latency gap between verification and high-budget generation, while Table 7 provides the exact measurements. The verifier itself introduces little wall-clock overhead under this configuration. Its mean latency is 0.67% of $B = 256$ generation latency, which is smaller than the 2.3% ratio obtained with the token-forward compute proxy used

Table 7: Measured stage-level wall-clock latency on one H100. Both models remain resident in memory, CUDA synchronization is applied, and warmup is excluded. Each configuration contains 38 timed runs.

Stage	Setting	Mean (s)	P50 (s)	P95 (s)	ms/step
Diffusion generation — LLaDA-8B					
LLaDA-8B	$B = 8$	0.186	0.182	0.224	23.24
LLaDA-8B	$B = 16$	0.367	0.357	0.430	22.92
LLaDA-8B	$B = 32$	0.732	0.715	0.861	22.87
LLaDA-8B	$B = 64$	1.465	1.429	1.723	22.88
LLaDA-8B	$B = 128$	2.927	2.862	3.450	22.87
LLaDA-8B	$B = 256$	5.849	5.707	6.893	22.85
Independent verification — Qwen2.5-7B					
Qwen2.5-7B	Verifier	**0.039**	**0.038**	**0.047**	–

Table 8: Policy-level latency of the executability cascade. Executable candidates terminate at B_0, whereas execution failures are independently regenerated at $B = 256$. Correct denotes the number of correct solutions among 664 tasks.

Policy	Accept	Mean (s)	P50 (s)	P95 (s)	Correct
Blind generation					
Blind $B = 256$	–	5.849	5.849	5.849	**273**
Executability-based selective escalation					
Exec. $B_0 = 32$	29.4%	4.863	6.581	6.581	238
Exec. $B_0 = 64$	54.5%	**4.125**	**1.465**	7.314	248
Exec. $B_0 = 128$	**75.9%**	4.336	2.927	8.776	269

in the main experiments. The two quantities measure different aspects of inference cost: token-forward accounting abstracts away hardware execution, whereas wall-clock latency benefits from the parallelism of a single verifier prefill while diffusion generation repeatedly evaluates the sequence across many denoising steps. We therefore retain token-forward computation for the main hardware-independent comparison and use Table 7 to characterize the deployment overhead of inserting the verifier stage.

Latency under selective escalation. Stage-level overhead alone does not determine end-to-end latency because a selective policy creates two execution paths. If the initial candidate is accepted, inference terminates after the first stage; otherwise, the request also pays for the high-budget regeneration stage. For an initial budget B_0, the corresponding mean latency can be written as

$$\mathbb{E}[T] = T_G(B_0) + T_{\text{gate}} + (1 - r_{\text{acc}})T_G(B_1), \tag{9}$$

where r_{acc} is the fraction of candidates terminating after the first stage. For the executability-based cascade considered here, T_{gate} is treated as zero and $B_1 = 256$. This baseline is separate from verification-guided inference and is included to isolate the latency effect of early termination and selective regeneration.

Table 8 reports the resulting policy-level latency for $B_0 \in \{32, 64, 128\}$. The P50 and P95 values in this table are induced by the two-path accounting in Eq. equation 9 using the measured mean stage costs from Table 7; they should therefore be distinguished from the empirical per-stage percentiles reported in Table 7.

Table 8 illustrates why average and tail latency need not move in the same direction under selective escalation. With $B_0 = 32$, only 29.4% of candidates terminate early, so the majority of requests still enter the expensive regeneration path; although the mean latency falls to 4.863 s, the median remains on the rejected path at 6.581 s. Increasing B_0 to 64 raises the early-termination rate to 54.5%, moving the median onto the fast path and producing the lowest mean latency, 4.125 s, but with only 248 correct solutions. The $B_0 = 128$ configuration provides a more favorable quality–latency trade-off: 75.9% of requests terminate after the 2.927-s first stage, reducing mean latency

Table 9: Effect of the deterministic validity stage at $B_0 = 64$ on HumanEval and MBPP. Verifier Calls denotes the fraction of all 664 moderate-budget candidates passed to Qwen2.5-7B-Instruct. Delivered Loud denotes overtly invalid moderate-budget candidates accepted without escalation.

Policy	Verifier Calls	Delivered Loud	Delivered Loud (%)
Verify-only	100.0%	18	2.7%
Validity + Verify	**54.5%**	**0**	**0.0%**

by 25.9% relative to blind $B = 256$, while producing 269 correct solutions compared with 273 for the blind baseline.

The same mechanism also explains the tail behavior in Table 8. A rejected $B_0 = 128$ request pays for both the $B_0 = 128$ stage and a full $B = 256$ regeneration, giving a rejected-path latency of 8.776 s. Because 24.1% of requests follow this path, the P95 latency also falls on the rejected branch. Selective escalation can therefore lower average and median latency without lowering the latency of every individual request. This distinction is particularly relevant for verification-guided inference: the independent verifier itself is inexpensive according to Table 7, but the end-to-end latency benefit ultimately depends on how often verification allows a request to terminate before expensive regeneration is invoked.

F Effect of the Deterministic Validity Stage

The main verification-guided pipeline applies a deterministic validity check before invoking the independent verifier. We additionally evaluate a *verify-only* alternative in which every moderate-budget candidate is passed directly to Qwen2.5-7B-Instruct, including overtly invalid outputs. This comparison isolates the contribution of the deterministic first-stage filter from the semantic verifier itself.

Table 9 reports the comparison at $B_0 = 64$ on the 664 HumanEval and MBPP tasks. Under the staged validity-then-verification pipeline, only 54.5% of candidates reach the independent verifier because the remaining candidates are rejected directly by the deterministic validity check. The verify-only alternative instead invokes the verifier for every candidate. Despite this additional verifier usage, it still accepts 18 overtly invalid outputs, corresponding to 2.7% of the full task set. The deterministic validity stage therefore serves two complementary purposes: it removes a large fraction of unnecessary verifier calls and prevents directly observable invalid moderate-budget outputs from being accepted without escalation. This ablation also clarifies the role of the two assurance stages. Independent verification is capable of rejecting most loud failures when they are presented to it directly, but using a language model to rediscover deterministically observable invalidity is unnecessary. The validity stage therefore handles failures for which a cheap task-specific signal is already available, while the independent verifier is reserved for candidates whose surface form is valid but whose semantic reliability remains uncertain.

G Prompt-Sensitivity Control

We evaluate whether the observed change in failure composition depends on the prompt format used for security-sensitive code generation. Using the same model (LLaDA-8B) and the same 24-task Python CWEval subset used in the LLaDA security evaluation, we compare whole-file generation with the anchor-splice protocol used in those experiments. The two settings therefore differ only in prompt construction. As shown in Table 10, both prompt formats exhibit a clear increase in silent share as denoising compute grows. Under whole-file generation, silent share rises from 0.0% at $B = 8$ to 35.0% at $B = 256$, while under anchor-splice prompting it increases from 0.0% to 26.8%. The absolute level is therefore prompt-sensitive, but the underlying shift toward surface-valid failures is preserved across both prompt formats.

Table 10: Prompt-sensitivity control on the same 24 CWEval tasks. Whole-file and anchor-splice prompting use the same model and task set. Correct and Silent Share are reported as fractions.

	Whole-File		**Anchor-Splice**	
B	Correct	Silent Share	Correct	Silent Share
8	0.000	0.000	0.000	0.000
16	0.000	0.047	0.000	0.000
32	0.016	0.122	0.000	0.005
64	0.016	0.153	0.073	0.124
128	0.073	0.247	0.094	0.213
256	0.167	**0.350**	0.182	**0.268**

Table 11: Verification of vulnerable outputs on the same 24 CWEval tasks. Recall measures the fraction of functionality-passing but vulnerable outputs rejected by the verifier. Confidence intervals are 95% task-level cluster bootstrap intervals over 10,000 replicates.

B	Vulnerability Recall	95% CI	$P(\text{vuln} \mid \text{accepted})$
Generic correctness verification			
64	18.2	[0.0, 80.0]	54.5
128	10.5	[0.0, 41.2]	82.9
256	21.1	[0.0, 63.2]	62.5
Security-aware verification			
64	**95.5**	[50.0, 100.0]	14.3
128	**94.7**	[76.5, 100.0]	25.0
256	**94.7**	[76.0, 100.0]	10.0

H Security-Aware Verification on CWEval

We further examine whether independent verification can detect security-sensitive silent failures using a separate generation set from the same 24-task Python CWEval subset. To isolate the effect of the verification objective, we apply both the generic correctness-verification prompt used in the main experiments and a security-aware prompt to the same generated candidates. Each task contributes eight candidates, and confidence intervals are obtained by cluster bootstrapping over the 24 underlying tasks rather than over individual generations. As shown in Table 11, the generic correctness prompt has low recall for functionality-passing vulnerabilities, detecting only 10.5%–21.1% across the three generation budgets. Explicitly conditioning verification on security raises recall to 94.7%–95.5%. The confidence intervals are necessarily wide because the analysis contains only 24 independent tasks, but the comparison indicates that the assurance objective itself matters: a verifier optimized for generic semantic correctness should not be assumed to detect domain-specific risks such as exploitability.