SPECULATIVE CORRECTION: DRAFT-THEN-REFINE DECODING FOR DIFFUSION LANGUAGE MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Diffusion language models (DLMs) can revise tokens bidirectionally, but standard decoding procedures often adapt them to left-to-right generation by producing text block by block. We show that existing pretrained DLMs can directly refine previously completed block-autoregressive responses, without additional training or architectural modification. This yields a simple *draft-then-refine* decoding pattern: a model first generates a complete response, then treats that response as an editable initialization for bidirectional refinement. We study both same-model correction, where one checkpoint drafts and refines, and small-to-large speculative correction, where separately released small and large checkpoints are composed without joint training or adaptation. Same-model correction improves the evaluated quality–latency trade-off: on a held-out GSM8K set, it raises accuracy from 0.789 to 0.916 with a 384-token budget while running $1.10\times$ faster, and on MATH-768 it matches the Flash score of 0.540 while running $1.52\times$ faster. Small-to-large correction retains much of the cheaper model's speed. On MATH-384, it scores 0.282 rather than 0.292 while running $2.32\times$ faster than Flash. On MBPP+, it raises the fraction of programs passing the expanded tests from 0.484 to 0.621 while running $1.15\times$ faster in standardized timing. Refinement is sparse: it changes only about five to six final tokens per response and fixes substantially more answers than it harms. Taken together, these results show that a completed response provides enough structure for a short refinement stage to make useful changes. Within this plug-and-play framework, same-model correction shows that an existing checkpoint can achieve a better quality–latency trade-off through decoding alone, while small-to-large correction can combine separately released checkpoints into a cascade that often adds an intermediate operating point between them. The success of this off-the-shelf cascade further motivates training small and large DLMs for complementary drafting and refinement roles as a dedicated architecture.

1 INTRODUCTION

Diffusion language models (DLMs) face a fundamental challenge in open-ended language generation: they are inherently bidirectional (Lou et al., 2024; Nie et al., 2025; Bie et al., 2026). Generation proceeds through denoising over an entire sequence simultaneously, allowing tokens to be revised under full bidirectional context. This makes DLM generation fundamentally distinct from autoregressive generation found in standard transformers, but also makes DLMs difficult to apply to standard language tasks, which are naturally posed as left-to-right generation problems. The prevailing solution is *block autoregressive diffusion* (block-AR diffusion), which generates text sequentially in blocks while applying bidirectional attention within each block. This compromise allows DLMs to approximate autoregressive (AR) processes (Xue et al., 2025) to better fit the language modeling paradigm.

As Figure 1 illustrates, this practical strategy constrains one of the core strengths of diffusion language models. By imposing an approximately autoregressive decoding regime on models designed for bidirectional revision, it limits their ability to revise globally across the sequence. This motivates decoding schemes that use bidirectional refinement more directly, rather than relying primarily on autoregressive approximation.

In this work, we study *draft-then-refine decoding* as a general *plug-and-play* inference pattern for diffusion language models. A first stage rapidly constructs a complete candidate response. A second stage applies full-sequence bidirectional refinement. In this framework, block-AR decoding is used

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.

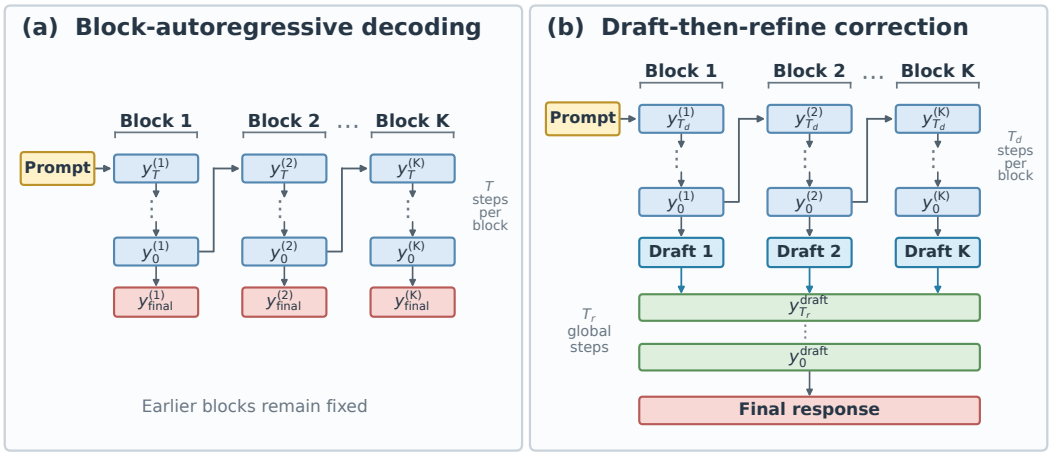

Figure 1: Block-autoregressive diffusion commits each block before generating the next. Draft-then-refine instead uses the completed block-AR response as an editable initialization for full-response correction.

not as a final generation procedure, but as a way to initialize a global corrector. The resulting architecture directly uses the ability of DLMs to review and revise earlier and later tokens jointly.

Success in this setting is not guaranteed. Although LLaDA2.1 (Bie et al., 2026) is trained for token-to-token editing, its editor is not trained specifically on completed block-autoregressive drafts produced under our drafting schedules, and the small-to-large setting additionally introduces cross-model mismatch. The draft must provide sufficient structure for the refiner to correct with only a few passes.

This architecture gives rise to two complementary questions. First, can an existing DLM checkpoint improve its own block-AR generations when computation is reallocated from purely sequential block denoising to complete-sequence drafting followed by global refinement? Second, can a smaller DLM provide a cheap but useful initialization that allows a larger DLM to recover much of its quality at substantially lower cost? We study both questions using the LLaDA2.1 family.

We find that both correction settings expand the evaluated quality–latency landscape across most settings. Same-model correction provides the clearest gains. On GSM8K-384, LLaDA2.1-flash→LLaDA2.1-flash raises accuracy from 0.789 to 0.916 while running $1.10\times$ faster than LLaDA2.1-flash. Small-to-large correction more often adds an intermediate operating point. On MATH-384, LLaDA2.1-mini→LLaDA2.1-flash scores 0.282 rather than 0.292 for LLaDA2.1-flash while running $2.32\times$ faster. The pattern is not universal. On MATH-768, LLaDA2.1-mini→LLaDA2.1-flash falls below both LLaDA2.1-mini and LLaDA2.1-flash, exposing the limitation of fixed-span refinement when the draft omits part of a long response.

This plug-and-play setting is deliberately stringent. The two checkpoints were not jointly trained or adapted for this cascade. LLaDA2.1-mini was not optimized for refinement-friendly drafting, and LLaDA2.1-flash was not trained to interpret Mini outputs as editable initializations. Positive results are therefore informative, while failures expose the limits of composing off-the-shelf checkpoints.

A mechanism study nevertheless shows that completed drafts provide effective initializations. Full-global refinement changes only a handful of final tokens and fixes substantially more answers than it harms, while alternative refinement schedules reach similar accuracy. This suggests that the completed draft matters more than any uniquely effective refinement schedule.

Our work is closely aligned with concurrent work on Diffusion in Diffusion (Ma et al., 2026), which also identifies draft-then-refine as a promising structure for block diffusion models. Diffusion in Diffusion realizes this idea through dedicated progressive refinement mechanisms and training within a single model. In contrast, we ask whether the same principle can be instantiated directly at inference time using existing pretrained DLMs, without additional training or architectural modification. Same-model correction provides evidence for the decoding primitive itself, while small-to-large correction extends it across model scales, showing that separately released DLM checkpoints can already form a useful inference-time cascade without joint adaptation.

Our main contributions are:

- We show that existing pretrained DLMs can refine completed block-autoregressive responses without additional training or architectural modification. In the same-model setting, this generally improves the evaluated quality–latency trade-off, showing that a second global pass can usefully reallocate a checkpoint's inference computation.
- We introduce *small-to-large speculative correction*, where a separately released small DLM produces the draft and a larger DLM refines the completed response. This allows existing checkpoints to be combined without joint training, distillation, or drafter–refiner adaptation, and often adds a faster intermediate operating point between the two models.
- We show that refinement is sparse and usually beneficial. It changes only a handful of final tokens, fixes substantially more answers than it harms, and performs similarly across several refinement schedules.

2 Related Work

Diffusion language models. Diffusion language models generate text by reversing a noising process over continuous representations (Li et al., 2022; Gong et al., 2023) or discrete token states (Hoogeboom et al., 2021; Austin et al., 2021; Campbell et al., 2022; Sahoo et al., 2024). Recent systems improve the practicality of discrete diffusion through score-entropy objectives (Lou et al., 2024) and large-scale masked modeling (Nie et al., 2025). LLaDA2.1 introduces token-to-token editing and describes Multiple Block Editing (MBE), which revisits earlier blocks during generation (Bie et al., 2026). Concurrent work by Mou et al. (2026) further develops MBE as a training-free procedure for revisiting previously decoded blocks. In contrast, our method performs refinement after a complete response is available. At the pinned revisions used here, the official release did not provide a runnable MBE inference path, so we use the released editing interface and refine only after a complete response is available.

Block diffusion decoding. Semi-autoregressive and blockwise diffusion make open-ended, prefix-conditioned generation practical by producing blocks sequentially while denoising bidirectionally within each block (Han et al., 2022; Arriola et al., 2025). This supports flexible-length generation and efficient mechanisms such as KV caching. Under the standard blockwise schedule, however, each finalized block becomes fixed prefix context and is not revisited after later context appears (Arriola et al., 2025). Our method retains block diffusion as a fast drafting procedure and adds full-response bidirectional correction after the complete draft is available.

Draft–then–refine diffusion. Coarse-to-fine generation is well established in diffusion modeling (Ho et al., 2022). The closest language work is Diffusion in Diffusion (Ma et al., 2026), which drafts with small blocks, selectively remasks low-confidence tokens, and refines with larger or global receptive fields. Its mix-scale training objective explicitly prepares a block diffusion model to operate at both drafting and refinement scales. We study the same broad principle under a different constraint: our procedure is plug-and-play and uses existing pretrained checkpoints without additional training or architectural modification. Same-model correction tests whether an off-the-shelf checkpoint can improve its own completed output. Beyond this setting, we introduce cross-checkpoint small-to-large correction: a separately released smaller DLM produces the draft, which is then revised by a larger DLM without joint cascade training, distillation, or drafter–refiner adaptation. Small and large DLMs have previously been combined through inference-time ensembling in David Helps Goliath (Han et al., 2024), but not, to our knowledge, through this completed-response drafting and correction cascade.

Speculative decoding. Autoregressive speculative decoding uses a cheap model to propose future tokens and a larger model to verify them in parallel (Leviathan et al., 2023; Chen et al., 2023). Variants use additional decoding heads, tree-structured proposals, or feature-level prediction (Cai et al., 2024; Miao et al., 2024; Li et al., 2024). DiffuSpec uses a pretrained DLM as a single-pass parallel drafter for an autoregressive verifier (Li et al., 2026), while DFlash trains a lightweight block-diffusion drafter conditioned on target-model features (Chen et al., 2026). Recent methods also adapt speculative verification to DLM decoding through self-drafting or masking-based mechanisms (Gao et al., 2025; Han et al., 2026; Cui et al., 2026). Concurrent xPress introduces a lightweight causal refiner to reconcile each diffusion draft block before autoregressive target verification (Wang et al., 2026). These methods use proposals or block refinements to accelerate a target decoding path.

Speculative Correction instead treats a completed response as an editable initialization and returns the refiner's output directly after bidirectional correction, without tokenwise acceptance.

3 Background: Diffusion Language Models

We briefly review the diffusion mechanism underlying DLMs to establish the notation and motivate the draft-then-refine approach.

3.1 Forward and Reverse Processes

Let $y \in \mathcal{V}^L$ denote a sequence of L tokens. A diffusion language model defines a forward process that gradually corrupts the clean sequence $y_0 = y$ into noise and a learned reverse process that reconstructs it:

$$y_0 \to y_1 \to \cdots \to y_T, \qquad y_T \to y_{T-1} \to \cdots \to y_0.$$

For conditional generation given a prompt x, standard DLM decoding begins from a highly corrupted or masked initialization y_T and iteratively recovers a sample from the learned conditional distribution over clean sequences. Our method instead begins refinement from a structured completed draft, motivating the connection in Section 3.3.

3.2 Block Autoregressive Diffusion

Standard fully bidirectional DLM generation is not directly aligned with left-to-right language tasks. Block-autoregressive diffusion provides a compromise by partitioning generation into blocks.

Let the response be divided into K blocks: $y = (y^{(1)}, y^{(2)}, \ldots, y^{(K)})$. Block-AR diffusion generates each block sequentially:

$$y^{(1)} \to y^{(2)} \to \cdots \to y^{(K)},$$

where each block $y^{(k)}$ is generated through the full diffusion process with bidirectional attention *within* that block. Critically, when generating block $y^{(k)}$, the model conditions on all previously generated blocks $(y^{(1)}, \ldots, y^{(k-1)})$ as context, but cannot attend to future blocks that have not yet been generated.

Within each block, the model applies T denoising steps:

$$y_T^{(k)} \to y_{T-1}^{(k)} \to \cdots \to y_0^{(k)},$$

where $y_T^{(k)}$ is initialized as noise and $y_0^{(k)}$ is the clean output for block k. Under an equal-cost-per-pass abstraction, the total computational cost for generating a sequence of K blocks with T steps per block is $K \times T$ denoising passes.

3.3 Connection to Draft-then-Refine

The draft-then-refine approach exploits a simple observation: if the initialization is already closer to the target distribution, fewer denoising steps may be required. Rather than starting from pure noise y_T, the refiner begins from a completed draft y_{draft} in which most of the response is already present, allowing a small refinement budget to focus on corrections. When the drafter and refiner use the same checkpoint, they share a model and tokenizer, although the completed blockwise draft may still differ from the intermediate states encountered during training. Small-to-large correction introduces additional cross-model mismatch. Our plug-and-play approach accepts this mismatch in exchange for composing existing checkpoints without joint training or architectural modification.

4 Methodology

We consider a prompt x and aim to generate a response y. A drafter first generates a complete candidate response, and a refiner then treats the generated span as an editable initialization.

4.1 Drafting

The drafter produces a response using block-autoregressive diffusion. Following the notation from Section 3, it partitions the response into K blocks and generates them sequentially with T_d denoising steps per block:

$$y^{\text{draft}} = D_{\theta_s}(x) = (y^{(1)}, y^{(2)}, \ldots, y^{(K)}).$$

Generation stops when the drafter produces an end-of-sequence token or reaches the maximum response length. The resulting token sequence—whether naturally terminated or capped—is the complete span presented to the refiner.

The drafting stage is designed to be inexpensive. It uses fewer denoising steps per block ($T_d < T$) than the standard block-AR baseline, whose cost is $K \times T$ denoising passes under an equal-cost abstraction. The drafter may be the same model as the refiner or a smaller model. In either case, the draft is not merely a proposal to accept or reject. It provides the structure that makes a short correction stage possible.

4.2 Bidirectional Refinement

The refiner receives the prompt and completed draft:

$$y^{\text{final}} = R_{\theta_l}(x, y^{\text{draft}}).$$

Prompt tokens remain fixed, while the generated span is editable. At each refinement pass, the model predicts every editable position using bidirectional context over the complete response. A proposed replacement is applied only when it differs from the current token and exceeds a confidence threshold. Refinement stops when no eligible change remains, when it produces a new end-of-sequence token, or when it reaches the maximum of T_r passes.

Because the refiner sees the complete candidate, an earlier token can change in light of later text and vice versa. The operation remains deliberately lightweight: it edits existing positions rather than regenerating the response from masks. It may shorten an answer by producing an earlier end-of-sequence token, but it cannot insert positions beyond the draft span. A truncated or structurally deficient draft can therefore limit what the second stage can recover. Exact token-handoff, end-of-sequence, and stopping rules are provided in Appendix A.

Under the same equal-cost abstraction, draft-then-refine uses at most $K \times T_d + T_r$ denoising passes rather than $K \times T$. This expression provides intuition rather than a latency model: the two stages may have different per-pass costs, full-span refinement differs from blockwise decoding, and early stopping changes the realized work. We therefore evaluate efficiency using synchronized model time, end-to-end generation time, and realized forward calls.

4.3 Correction Configurations

We study two general ways to assign the drafting and refinement roles.

Same-model correction. The same checkpoint serves as both drafter and refiner, so $\theta_s = \theta_l$. This setting removes cross-model mismatch and asks whether reallocating computation from blockwise generation to completed-response correction can improve the model's own output.

Small-to-large speculative correction. A smaller checkpoint drafts and a larger checkpoint refines, so $\theta_s \neq \theta_l$. This setting asks whether the inexpensive first stage can retain enough structure for a short large-model correction pass. Unlike autoregressive speculative decoding, the larger model does not verify proposals sequentially. It edits the completed response under bidirectional context.

5 Experimental Protocol

Our experiments ask three questions. Does a completed-response correction stage improve generation by an existing checkpoint? Can a smaller drafter preserve most of its speed while a larger refiner recovers quality? Finally, what does the refiner actually change, and when does the approach fail?

5.1 Models

We use LLaDA2.1-mini and LLaDA2.1-flash from the LLaDA2.1 family (Bie et al., 2026), without training, distillation, or fine-tuning. Both checkpoints support block-autoregressive generation and token-to-token editing over an existing response. All block-AR baselines use the official generation path available at the pinned revisions; this path does not reopen previously completed blocks through MBE. They also share a tokenizer, which allows us to transfer a draft without changing its tokenization. We instantiate same-model correction as Flash→Flash and small-to-large speculative correction as Mini→Flash.

LLaDA2.1 provides the two model sizes and editable diffusion interface needed to test both settings within one family. At the time of writing, we are not aware of another publicly available DLM family that provides both multiple model scales and a compatible token-editing interface for a controlled small-to-large evaluation. This availability constraint motivates our focus on LLaDA2.1.

5.2 Tasks and Evaluation Sets

We evaluate MATH, GSM8K, and MBPP+ because they require multi-step reasoning or structured generation, where revision can change the correctness of the final response. A suffix such as 384 or 768 gives the maximum number of generated response tokens, excluding the prompt.

For MATH, configurations are selected on development data disjoint from the held-out 500-problem evaluation set. GSM8K uses a held-out set of 95 problems, and MBPP+ uses 153 content-disjoint tasks. We fix the evaluation sets and decoding configurations before examining their outputs. Exact construction and overlap accounting appear in Appendix B.[1]

5.3 Protocol

Each primary evaluation compares four methods: block-autoregressive generation with Mini, block-autoregressive generation with Flash, Mini→Flash small-to-large correction, and Flash→Flash same-model correction. Flash is the principal baseline because it is the stronger single-stage model that correction aims to improve or accelerate. Mini shows whether the second stage improves the inexpensive draft rather than merely preserving it.

Configuration selection. Configuration selection follows staged, method-specific development searches rather than an exhaustive search for every final panel. During exploratory development, we sweep both standalone decoders over broad ranges of block size, denoising budget, and confidence threshold, and similarly explore correction configurations over drafting and refinement parameters. Under the final protocol, we then re-screen selected regions on development data disjoint from the held-out evaluation sets, including four Mini configurations specified in advance and a 12-configuration Flash grid on MATH-384, together with separate Mini→Flash and Flash→Flash searches. Later longer-budget studies use regions identified by these searches to guide narrower follow-up experiments rather than repeating the full search space. All primary configurations are frozen before final evaluation. This staged protocol provides a fair comparison under finite compute. Appendix C gives the detailed search coverage and frozen configurations.

For the frozen configurations, we evaluate quality using final-answer accuracy on MATH and GSM8K and pass@1 on MBPP+, where Base uses the original tests and Plus uses the stricter expanded tests. We report 95% problem-level bootstrap confidence intervals. Quality is measured on the full evaluation sets, while latency is measured in isolated repeated runs of the same configurations. Full scoring, statistical, and systems protocols appear in Appendices B and D.

6 Experiments

6.1 Quality and Quality–Latency Trade-offs

In this section, we conduct a broad evaluation across mathematical reasoning, grade-school arithmetic, and code generation. Table 1 compares the frozen development-selected configuration for each method. Within each row, all methods use the same problems and output budget.

Same-model correction provides the clearest evidence that a completed draft can improve generation without additional training. On GSM8K, Flash→Flash raises Flash accuracy from 0.789 to 0.916 at 384 tokens and to 0.937 at 512 tokens. The timed 384-token configuration is also $1.10\times$ faster than Flash. On MATH-768, it preserves the Flash score of 0.540 while running $1.52\times$ faster. On MBPP+, it raises the expanded-test score from 0.484 to 0.660, although the corresponding timed configuration is $0.89\times$ as fast as Flash. MATH-384 instead yields a faster but slightly less accurate point, scoring 0.272 rather than 0.292 while running $1.41\times$ faster.

Small-to-large correction more often traces an intermediate operating point between Mini and Flash. On MATH-384, Mini→Flash scores 0.282 rather than 0.292 for Flash while running $2.32\times$ faster.

[1] Additional results on development-exposed evaluation pools are reported separately in Appendix I and are not included in the primary evidence.

Table 1: Quality on held-out evaluation sets. Values are scores with 95% problem-level bootstrap confidence intervals. MBPP+ reports Base and Plus pass@1.

Task	n	Mini	Flash	Mini→Flash	Flash→Flash
MATH-384	500	.266 [.228,.304]	.292 [.252,.332]	.282 [.242,.322]	.272 [.234,.312]
MATH-768	500	.400 [.358,.444]	.540 [.496,.582]	.376 [.334,.418]	.540 [.496,.584]
GSM8K-384	95	.653 [.558,.747]	.789 [.705,.863]	.705 [.611,.789]	.916 [.853,.968]
GSM8K-512	95	.653 [.558,.747]	.789 [.705,.863]	.705 [.611,.789]	.937 [.884,.979]
MBPP+-384 Base	153	.510 [.431,.588]	.575 [.497,.654]	.745 [.673,.810]	.778 [.712,.843]
MBPP+-384 Plus	153	.412 [.333,.490]	.484 [.405,.562]	.621 [.542,.699]	.660 [.582,.732]

On GSM8K-384, it improves upon Mini from 0.653 to 0.705 while remaining $1.29\times$ faster than Flash, which scores 0.789. On MBPP+, it exceeds Flash under the expanded tests, scoring 0.621 rather than 0.484 while running $1.15\times$ faster.

To isolate the effect of refinement, we also score each correction run before and after the refinement stage. This holds the generated draft fixed, so any change in quality comes from refinement rather than from a different drafting configuration. For both Mini→Flash and Flash→Flash, refinement generally improves the exact draft across the evaluated settings. Appendix C reports the corresponding draft-only results.

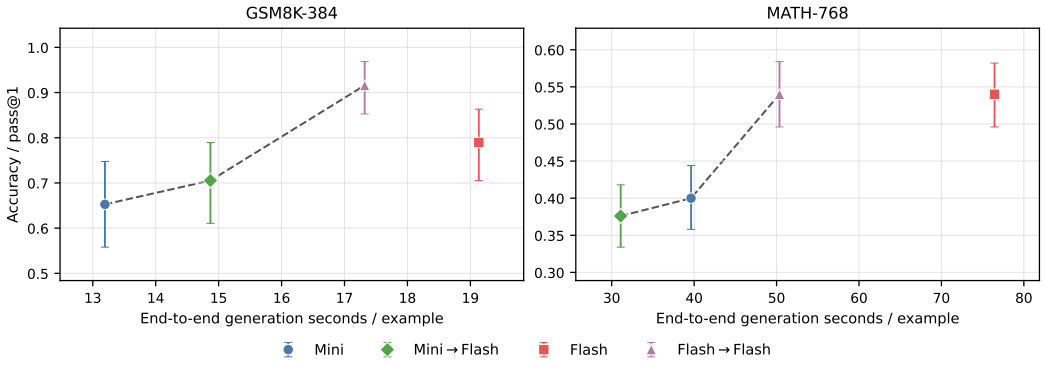

Figure 2: Selected quality–latency trade-offs for the timed configurations. Error bars show 95% bootstrap confidence intervals. Dashed lines connect nondominated methods. The remaining panels appear in Appendix E.

Figure 2 summarizes these two roles of correction. Same-model correction generally improves the quality–latency trade-off through better quality, lower latency, or both, while small-to-large correction often adds a faster intermediate operating point. This pattern repeats across most settings tested, but is not universal. On MATH-768, Mini→Flash remains $2.46\times$ faster than Flash but scores 0.376, below Mini at 0.400 and Flash at 0.540. Because the refiner edits a fixed span and cannot add a missing continuation, an incomplete long-form draft can limit what it recovers. Detailed length and termination results appear in Appendix F.

6.2 Systems Efficiency

Quality alone does not determine whether correction provides a useful operating point, so the preceding section considers quality together with latency. Here, we examine generation cost more directly through separate, controlled timing runs. Table 2 measures the same four methods under a standardized deployment and repeated-measurement protocol. Synchronized model time and end-to-end generation time are close, indicating that the measured differences primarily reflect model computation rather than substantial non-model overhead.

Mini→Flash retains much of Mini's latency advantage, running $2.32\times$, $1.29\times$, and $1.15\times$ faster than Flash on MATH-384, GSM8K-384, and MBPP-384, respectively. Flash→Flash is $1.41\times$, $1.52\times$, and $1.10\times$ faster on MATH-384, MATH-768, and GSM8K-384, while its MBPP gain costs latency. Requested step counts alone do not determine runtime because editing may stop early and full-response passes differ in cost from blockwise passes. Both checkpoints remain resident during measurement, requiring 123.7–131.0 GiB of reserved memory.

Table 2: Standardized systems measurements under the isolated two-GPU deployment. Speedup is relative to Flash end-to-end time; forward calls, token counts, refinement spans, and edits are per-example means.

Panel	Method	Sync s	E2E s	Speedup	Forwards	Draft tok.	Refine span	Edits	Peak GiB
MATH-384	Mini	19.79	19.85	2.49×	57.3	362.2	0.0	0.0	126.4
	Flash	49.39	49.47	1.00×	81.5	359.1	0.0	0.0	126.4
	Mini→Flash	21.14	21.30	2.32×	59.3	362.2	362.2	7.5	126.4
	Flash→Flash	34.98	35.16	1.41×	57.8	360.4	360.3	16.1	126.4
MATH-768	Mini	39.58	39.63	1.93×	119.8	552.8	0.0	0.0	131.0
	Flash	76.37	76.45	1.00×	126.8	545.7	0.0	0.0	131.0
	Mini→Flash	30.94	31.09	2.46×	93.1	514.4	514.4	17.7	131.0
	Flash→Flash	50.19	50.35	1.52×	84.2	557.2	557.1	44.4	131.0
GSM8K-384	Mini	13.14	13.19	1.45×	39.7	215.0	0.0	0.0	123.7
	Flash	19.05	19.13	1.00×	32.9	173.9	0.0	0.0	123.7
	Mini→Flash	14.72	14.87	1.29×	42.3	215.0	214.9	6.0	123.7
	Flash→Flash	17.15	17.32	1.10×	29.8	178.8	178.7	9.9	123.7
MBPP-384	Mini	5.03	5.08	1.44×	15.4	95.5	0.0	0.0	130.7
	Flash	7.21	7.29	1.00×	12.5	88.4	0.0	0.0	130.7
	Mini→Flash	6.17	6.31	1.15×	17.4	95.5	95.3	4.4	130.7
	Flash→Flash	8.04	8.21	0.89×	13.9	91.9	91.9	3.0	130.7

6.3 Completed Drafts Enable Sparse Correction

To understand how a short refinement stage acts on a completed response, we conduct a paired mechanism study on 300 GSM8K-384 and 300 MATH-384 problems. For each problem, four refinement schedules begin from exactly the same Mini draft. Local refinement edits independent 64-token chunks. Context-preserving blockwise refinement keeps the complete draft visible while making one block editable at a time, and hybrid refinement follows these blockwise passes with a full-global pass. Full-global refinement makes the entire generated span editable at once. A draft-only arm establishes the starting point, while a mask-only arm tests whether the same small refinement budget can produce a useful response without the completed draft. *Answer changed* counts examples in which the extracted answer differs from the draft. *Other text changed* counts examples in which wording, formatting, or reasoning text changes while the extracted answer remains fixed.

Table 3: Paired refinement-schedule study on 300 examples per task. Draft-conditioned arms share Mini drafts. Edits and final changes are per-example means; answer changes, other changes, and fix/harm are problem counts relative to draft-only.

Task	Arm	Score [95% CI]	Sync s	Forwards	Edits	Final Δ	Answer Δ	Other Δ	Fix / harm
GSM8K	Draft only	.733 [.680,.783]	12.36	38.2	0.00	0.00	0	0	0 / 0
	Mask only	.000 [.000,.000]	1.25	2.6	12.49	5.04	300	0	0 / 220
	Local chunks	.767 [.720,.813]	15.67	44.2	5.01	4.49	18	258	12 / 2
	Context blockwise	.770 [.723,.817]	16.20	44.7	5.81	4.65	20	257	13 / 2
	Hybrid	.770 [.723,.817]	16.33	44.9	5.98	4.92	20	256	13 / 2
	Full global	.770 [.720,.817]	13.89	40.8	6.14	4.86	20	256	13 / 2
MATH	Draft only	.223 [.177,.270]	18.00	53.0	0.00	0.00	0	0	0 / 0
	Mask only	.000 [.000,.000]	0.97	2.0	13.56	11.71	290	10	0 / 67
	Local chunks	.237 [.190,.287]	22.28	60.8	6.46	5.92	9	270	5 / 1
	Context blockwise	.240 [.193,.290]	22.94	61.2	6.67	5.58	8	269	6 / 1
	Hybrid	.243 [.197,.293]	22.02	59.7	6.85	6.85	9	271	7 / 1
	Full global	.243 [.197,.293]	19.17	55.0	7.05	5.73	9	271	7 / 1

The completed draft is essential under this limited refinement budget. Mask-only initialization produces no correct answers, while every draft-conditioned schedule raises the observed score above the draft-only result. The mask-only arm is an initialization diagnostic rather than a comparison with fully budgeted generation from masks.

Among the draft-conditioned schedules, accuracy varies little. Scores range from 0.767 to 0.770 on GSM8K and from 0.237 to 0.243 on MATH. Full-global refinement remains the least expensive correction arm, requiring 13.89 synchronized seconds on GSM8K and 19.17 on MATH. These results suggest that the completed draft matters more than the attention schedule. They support full-global refinement as an efficient default, but not as uniquely responsible for the quality improvement.

Refinement makes sparse changes to the final output. Full-global refinement performs 6.14 edit events per GSM8K response and 7.05 per MATH response, but only 4.86 and 5.73 changes remain in the respective final outputs because some positions are edited more than once. It also inserts EOS in 41 GSM8K and 11 MATH examples. Despite changing some non-answer text in most responses, it changes the extracted answer in only 20/300 GSM8K cases and 9/300 MATH cases. Relative to draft-only correctness, these outputs contain 13 fixes versus 2 harms on GSM8K and 7 fixes versus 1 harm on MATH.

The primary held-out evaluations show a similar favorable asymmetry at the method level, although these comparisons evaluate complete methods rather than following the same draft. On GSM8K-384 and GSM8K-512, Flash→Flash produces 14 favorable versus 2 unfavorable correctness transitions and 15 versus 1 relative to Flash. On the MBPP+ Base and Plus tests, Mini→Flash produces 34 favorable versus 8 unfavorable transitions and 33 versus 12 relative to Flash. Taken together, these results show that correction preserves most of the completed draft while making a small number of disproportionately useful changes.

7 Discussion and Future Work

The two correction settings reveal a common principle. A completed response can be more useful as an editable starting point than as a final commitment. Same-model correction isolates this decoding effect within one checkpoint, while small-to-large correction shows that it can survive when the draft is produced more cheaply. The latter often adds an operating point between Mini and Flash, but its failures expose the central boundary of the current method. Fixed-span refinement can repair an existing response, but it cannot recover a continuation that the drafter never produced.

The mechanism study sharpens this interpretation. Similar accuracy across several refinement schedules suggests that the completed draft matters more than the precise schedule used to refine it. The sparse and predominantly beneficial changes further indicate that correction works by selectively revising an already structured response rather than regenerating it.

The plug-and-play design avoids additional training, but it also has practical costs. Mini and Flash are simultaneously resident in the controlled two-GPU deployment, and the observed memory use leaves little room for both checkpoints on a single device. The evidence is also limited to one DLM family, two compatible checkpoints, and single-example generation under one deployment. We therefore conclude that speculative correction improves the quality–latency trade-off across several distinct settings, not that it universally dominates every baseline or deployment.

A natural next step is to train models specifically for the drafting and refinement roles. A drafter could be optimized not only for standalone accuracy, but also for refinability, reliable termination, global structure, and explicit representation of uncertainty. The refiner could likewise be adapted to drafts produced by the smaller model rather than only to states from its own noising process. Separate role specialization or joint training could therefore reduce the mismatch that limits off-the-shelf small-to-large correction.

Long responses also motivate hierarchical correction. A drafter could generate shorter segments, each segment could be corrected before adjacent segments are combined, and progressively larger spans could then receive global refinement. This follows the multi-scale motivation of Diffusion in Diffusion (Ma et al., 2026), but assigns drafting and refinement to separately released checkpoints. A simpler extension would reserve an editable masked tail so that the refiner can continue an incomplete response. Both approaches would relax the current fixed-span constraint.

8 Conclusion

Diffusion language models should be decoded in a way that uses their defining capability: bidirectional revision. Block-autoregressive diffusion is practical, but it commits to earlier text before the complete response is available. Draft-then-refine decoding offers a simple alternative. It first constructs a full response and then gives the model an opportunity to correct that response as a whole. Same-model correction shows that this opportunity can improve an existing checkpoint without training or architectural modification. Small-to-large speculative correction goes further by showing that separately released checkpoints can be composed into new quality–latency operating points without joint adaptation.

Ethics Statement

This work studies inference-time decoding methods using publicly available models and benchmark datasets. It does not involve human subjects, collection of personal data, or deployment in high-risk settings. We are not aware of any additional ethical concerns specific to this work.

Reproducibility statement

The appendices specify the token-handoff and stopping rules, evaluation-set construction, scoring procedures, frozen configurations and model revisions, statistical analysis, systems protocol, and evidence boundaries. The codebase will be released publicly.

AI use statement

We used generative AI tools to assist with experimental implementation, provide feedback on experimental methodology and interpretation of results, assist with literature retrieval and reference checking, and assist with drafting and editing the manuscript. The authors reviewed all AI-assisted work and take responsibility for the reported methods, results, interpretations, references, and final manuscript. Experiment execution and all final scientific decisions were supervised by the authors.

References

Marianne Arriola, Aaron Gokaslan, Justin T. Chiu, Zhihan Yang, Zhixuan Qi, Jiaqi Han, Subham Sekhar Sahoo, and Volodymyr Kuleshov. Block diffusion: Interpolating between autoregressive and diffusion language models. *arXiv preprint arXiv:2503.09573*, 2025.

Jacob Austin, Daniel D. Johnson, Jonathan Ho, Daniel Tarlow, and Rianne van den Berg. Structured denoising diffusion models in discrete state-spaces. In *Advances in Neural Information Processing Systems*, 2021.

Tiwei Bie et al. LLaDA2.1: Speeding up text diffusion via token editing. *arXiv preprint arXiv:2602.08676*, 2026.

Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D. Lee, Deming Chen, and Tri Dao. Medusa: Simple llm inference acceleration framework with multiple decoding heads, 2024. URL `https://arxiv.org/abs/2401.10774`.

Andrew Campbell, Joe Benton, Valentin De Bortoli, Tom Rainforth, George Deligiannidis, and Arnaud Doucet. A continuous time framework for discrete denoising models. In *Advances in Neural Information Processing Systems*, 2022.

Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318*, 2023.

Jian Chen, Yesheng Liang, and Zhijian Liu. DFlash: Block diffusion for flash speculative decoding. *arXiv preprint arXiv:2602.06036*, 2026.

Junxia Cui, Haotian Ye, Runchu Tian, Hongcan Guo, Jinya Jiang, Haoru Li, Chaojie Ren, Yiming Huang, Kaijie Zhu, Zhongkai Yu, Kun Zhou, and Jingbo Shang. SimSD: Simple speculative decoding in diffusion language models. *arXiv preprint arXiv:2606.02544*, 2026.

Yifeng Gao, Ziang Ji, Yuxuan Wang, Biqing Qi, Hanlin Xu, and Linfeng Zhang. Self speculative decoding for diffusion large language models. *arXiv preprint arXiv:2510.04147*, 2025.

Shansan Gong, Mukai Li, Jiangtao Feng, Zhiyong Wu, and Lingpeng Kong. DiffuSeq: Sequence to sequence text generation with diffusion models. In *International Conference on Learning Representations*, 2023.

Ligong Han, Hao Wang, Han Gao, Kai Xu, and Akash Srivastava. S2D2: Fast decoding for diffusion LLMs via training-free self-speculation. *arXiv preprint arXiv:2603.25702*, 2026.

Xiaochuang Han, Sachin Kumar, and Yulia Tsvetkov. SSD-LM: Semi-autoregressive simplex-based diffusion language model for text generation and modular control. *arXiv preprint arXiv:2210.17432*, 2022.

Xiaochuang Han, Sachin Kumar, Yulia Tsvetkov, and Marjan Ghazvininejad. David helps goliath: Inference-time collaboration between small specialized and large general diffusion LMs. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 8385–8400. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.naacl-long.464.

Jonathan Ho, Chitwan Saharia, William Chan, David J. Fleet, Mohammad Norouzi, and Tim Salimans. Cascaded diffusion models for high fidelity image generation. *Journal of Machine Learning Research*, 23(47):1–33, 2022.

Emiel Hoogeboom, Didrik Nielsen, Priyank Jaini, Patrick Forré, and Max Welling. Argmax flows and multinomial diffusion: Learning categorical distributions. In *Advances in Neural Information Processing Systems*, 2021.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, 2023.

Guanghao Li, Zhihui Fu, Min Fang, Qibin Zhao, Ming Tang, Chun Yuan, and Jun Wang. DiffuSpec: Unlocking diffusion language models for speculative decoding. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 20896–20910. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.findings-acl.1048.

Xiang Lisa Li, John Thickstun, Ishaan Gulrajani, Percy Liang, and Tatsunori B. Hashimoto. Diffusion-LM improves controllable text generation. In *Advances in Neural Information Processing Systems*, 2022.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. EAGLE: Speculative sampling requires rethinking feature uncertainty. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 28935–28948. PMLR, 2024. URL https://proceedings.mlr.press/v235/li24bt.html.

Aaron Lou, Chenlin Meng, and Stefano Ermon. Discrete diffusion modeling by estimating the ratios of the data distribution. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 32819–32848. PMLR, 2024. URL https://proceedings.mlr.press/v235/lou24a.html.

Linrui Ma, Yufei Cui, Kai Han, and Yunhe Wang. Diffusion In Diffusion: Reclaiming global coherence in semi-autoregressive diffusion. *arXiv preprint arXiv:2601.13599v2*, 2026. URL https://arxiv.org/abs/2601.13599v2.

Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, Chunan Shi, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. Specinfer: Accelerating large language model serving with tree-based speculative inference and verification. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, pp. 932–949. ACM, April 2024. doi: 10.1145/3620666.3651335. URL http://dx.doi.org/10.1145/3620666.3651335.

Xingyu Mou, Zijin Huang, Tianze Zhang, Yuxin Ma, Lanning Wei, Zengfeng Huang, Da Zheng, and Lun Du. Beyond block boundaries: Multi-block editing for diffusion large language models. *arXiv preprint arXiv:2607.22663*, 2026.

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.

Subham Sekhar Sahoo, Marianne Arriola, Yair Schiff, Aaron Gokaslan, Edgar Marroquin, Justin T Chiu, Alexander Rush, and Volodymyr Kuleshov. Simple and effective masked diffusion language models, 2024. URL https://arxiv.org/abs/2406.07524.

Zheng Wang, Davis Wertheimer, Yu Chin Fabian Lim, Mudhakar Srivatsa, Raghu K. Ganti, Minjia Zhang, and Naigang Wang. xpress: Parallel refinement for diffusion drafters in speculative decoding. *arXiv preprint arXiv:2608.02438*, 2026. URL https://arxiv.org/abs/2608.02438.

Shuchen Xue, Tianyu Xie, Tianyang Hu, Zijin Feng, Jiacheng Sun, Kenji Kawaguchi, Zhenguo Li, and Zhi-Ming Ma. Any-order gpt as masked diffusion model: Decoupling formulation and architecture, 2025. URL https://arxiv.org/abs/2506.19935.

The appendix provides implementation and evaluation details, supporting analyses, and comparisons with earlier results.

A Token Handoff and Stopping Details

The drafter passes its output token IDs directly to the refiner. In the small-to-large setting, we first verify that the two tokenizers have compatible token-ID spaces and stop if a lossless handoff cannot be established. Direct transfer avoids changing boundary or special-token behavior through an intermediate decode-and-tokenize operation. Prompt tokens remain fixed throughout refinement.

If the draft contains an end-of-sequence token, we discard positions after its first occurrence and keep that token fixed. If the draft contains no end-of-sequence token, the full generated span remains editable. On each pass, the refiner predicts all editable positions and applies only replacements that differ from the current token and exceed the configured confidence threshold. Refinement stops when no such edit remains, when an edit produces a new end-of-sequence token, or when the maximum number of passes is reached.

The maximum refinement count is therefore an upper bound rather than a guarantee of realized computation. For every example, we record the realized forward calls, edit events, final token changes, end-of-sequence events, and final refinement span. Refinement never inserts positions beyond the transferred span. It can shorten a response by inserting an earlier end-of-sequence token, but it cannot reconstruct an arbitrarily long missing continuation.

B Datasets and Evaluation

Locked MATH. MATH uses the 500 examples from the `HuggingFaceH4/MATH-500` evaluation split. Development examples and examples used in earlier project experiments were recorded before the final protocol was frozen. The 500-example quality pool used in Table 1 has no overlap with those development sets. The same problem IDs are evaluated at the 384- and 768-token budgets. Each retained configuration produces one temperature-zero output per problem.

Project-untouched GSM8K. GSM8K is pinned to dataset revision `740312add88f781978c0658806c59bc2815b9866`. Before fixing the evaluation set, we identified all canonical test IDs present in saved outputs from earlier experiments. Of 1,319 test problems, 1,224 had appeared in those outputs and 95 had not. We evaluate all 95 remaining problems once at 384 and 512 tokens with the four configurations fixed in advance. This establishes independence from the project's experiment history, not from model pretraining.

Project-untouched MBPP+. The pinned EvalPlus 0.3.1 release contains 378 MBPP+ v0.2.0 tasks. Earlier experiments used 257 MBPP IDs, of which 224 overlap MBPP+. We also exclude `Mbpp/569`, whose normalized natural-language problem duplicates historical `mbpp-104` despite having a different ID. The evaluation set contains all 153 eligible content-disjoint tasks; there is no outcome-based subsampling. Generation uses one temperature-zero completion per method. Final scoring uses official EvalPlus sanitization and reports Base and Plus pass@1. The reported scores come from EvalPlus evaluation, rather than checks performed during generation.

Historical comparisons and mechanism study. Historical comparisons reuse previously evaluated GSM8K, MATH, MBPP, HumanEval, and synthetic examples with the corrected implementation. The mechanism study uses 300 MATH and 300 GSM8K problems selected before the six-arm outputs were examined. These additional evaluations support comparisons with earlier results or paired mechanism analyses; they are excluded from the primary quality estimates.

Invalid outputs. All benchmark examples remain in the denominator. Missing numeric answers, malformed symbolic answers, syntax errors, failed tests, subprocess errors, and timeouts count as incorrect. No result table filters generations after observing their output.

Metrics. For GSM8K and MATH, we report final-answer accuracy after task-specific extraction and normalization. GSM8K uses the last extractable number. MATH prefers the last boxed expression or explicit final-answer marker and otherwise applies normalized numeric or symbolic matching. For MBPP+, Base pass@1 uses the original MBPP tests and Plus pass@1 uses the expanded MBPP+ tests. We use EvalPlus 0.3.1 with MBPP+ v0.2.0 and official sanitization. The independent unit is the problem ID, with one temperature-zero output per method. Confidence intervals and paired comparisons are described in Appendix D.

C Configuration Search, Frozen Configurations, and Model Revisions

Configuration search follows a staged development protocol. Across exploratory tasks and output lengths, we evaluated both standalone decoders over broad ranges of block size, denoising budget, and confidence threshold, together with separate searches over drafting and refinement parameters for the correction methods. The standalone Mini experiments collectively covered 23 distinct schedules, spanning block sizes 16–128, denoising budgets 4–32, and confidence thresholds 0.20–0.50. These schedules were distributed across exploratory settings rather than evaluated as a single Cartesian grid on one common development set.

Under the final protocol, we re-screened promising regions on development data disjoint from the held-out evaluation sets before freezing the primary configurations. On MATH-384, this included four Mini configurations specified in advance, a 12-configuration Flash grid, and 13 Mini→Flash configurations varying the draft schedule, refinement budget, and confidence threshold. Flash→Flash was subsequently examined through seven additional configurations. For MATH-768, regions identified by the earlier searches guided narrower follow-up experiments, including five Mini→Flash variants. A full Cartesian search across every method, task, and output length was not feasible under the available compute, so later searches focused on promising regions identified at earlier stages. This protocol preserves the separation between development-based configuration selection and held-out evaluation, but does not establish an exhaustive quality–latency frontier for every method and panel.

All experiments use the public LLaDA2.1 checkpoints without training, distillation, or fine-tuning. The frozen model revisions are:

- LLaDA2.1-mini: `20e64e2ad21644d0e5248586ed9c942cdd45de0f`
- LLaDA2.1-flash: `2bf95e86ade33c1adb5c1e223b7db2a76dc3bdd6`

We use the shorthand b for block length, s for block denoising steps, and t for confidence threshold. For cascades, d and r distinguish draft and refinement parameters. Table 4 lists the configurations used for the main quality table and frontier figure.

The primary MATH-384 Mini→Flash quality–latency comparison uses the configuration scoring 0.282, for which both quality and standardized timing were measured. The configuration scoring 0.284 is reported as an additional quality-only result.

The MATH-384 evaluation also includes a Mini→Flash schedule $d(b64, s16, t.25), r(1, t.20)$ scoring 0.270 and Flash schedules $b64, s24, t.50$ and $b64, s8, t.35$ scoring 0.282 and 0.268. Together with the secondary 0.284 Mini→Flash configuration in Table 4, all eight configurations were fixed before final generation.

Draft-only controls. We score the exact pre-refinement draft stored by each correction run using the same task scorers as the final output. For Mini→Flash, refinement improves the exact draft on every primary panel. On MATH-384, accuracy rises from 0.266 to 0.282, with 11 fixes and 3 harms, and on MATH-768 from 0.352 to 0.376, with 16 fixes and 4 harms. On GSM8K, accuracy rises from 0.653 to 0.705 at both 384 and 512 tokens, with 6 fixes and 1 harm at each length. On MBPP+, Base pass@1 rises from 0.510 to 0.745, with 41 fixes and 5 harms, while Plus pass@1 rises from 0.412 to 0.621, with 35 fixes and 3 harms. The Mini→Flash drafts are identical to the standalone Mini outputs wherever the drafting configuration is shared. MATH-768 uses a different Mini drafting threshold, so its exact draft score is 0.352 rather than the separately configured Mini baseline of 0.400.

Table 4: Frozen configurations used in the primary quality and timing results, with secondary MATH-384 configurations retained for completeness.

Panel / method	Frozen schedule	Role
MATH-384 Mini	$b64, s8, t.40$	quality and timing
MATH-384 Flash	$b32, s16, t.50$	primary Flash; quality and timing
MATH-384 Mini→Flash	$d(b64, s8, t.40), r(2, t.20)$	primary quality and timing, score .282
MATH-384 Mini→Flash	$d(b64, s8, t.40), r(4, t.35)$	secondary quality configuration, score .284
MATH-384 Flash→Flash	$d(b64, s16, t.25), r(4, t.20)$	quality and timing
MATH-768 Mini	$b64, s8, t.40$	quality and timing
MATH-768 Flash	$b32, s16, t.50$	quality and timing
MATH-768 Mini→Flash	$d(b64, s8, t.25), r(1, t.20)$	quality and timing
MATH-768 Flash→Flash	$d(b64, s16, t.25), r(4, t.20)$	quality and timing
GSM8K Mini	$b64, s16, t.40$	384/512 quality; 384 timing
GSM8K Flash	$b64, s32, t.50$	384/512 quality; 384 timing
GSM8K Mini→Flash	$d(b64, s16, t.40), r(4, t.30)$	384/512 quality; 384 timing
GSM8K Flash→Flash	$d(b64, s16, t.25), r(4, t.20)$	384/512 quality; 384 timing
MBPP+ Mini	$b64, s16, t.40$	384 quality and timing
MBPP+ Flash	$b128, s32, t.50$	384 quality and timing
MBPP+ Mini→Flash	$d(b64, s16, t.40), r(4, t.30)$	384 quality and timing
MBPP+ Flash→Flash	$d(b64, s8, t.40), r(4, t.20)$	384 quality and timing

For Flash→Flash, refinement raises the exact draft from 0.234 to 0.272 on MATH-384, with 22 fixes and 3 harms; from 0.462 to 0.540 on MATH-768, with 48 fixes and 9 harms; from 0.705 to 0.916 on GSM8K-384, with 20 fixes and no harms; and from 0.726 to 0.937 on GSM8K-512, again with 20 fixes and no harms. On the corresponding timing panels, refinement adds approximately 2.11 synchronized seconds on MATH-384, 2.14 seconds on MATH-768, and 1.70 seconds on GSM8K-384. These paired comparisons hold the drafting configuration and realized draft fixed, isolating the incremental effect of refinement for each frozen cascade configuration.

D Statistical and Systems Protocol

The independent unit for quality evaluation is the problem ID. Each problem contributes one output per method, and outputs are aligned by problem ID for every primary comparison. We compute 95% confidence intervals for marginal scores and paired score differences using 20,000 percentile-bootstrap resamples over problem IDs. Fixes count problems where the reference is wrong and the candidate is correct; harms count the reverse.

The systems protocol measures each method after ten warmup examples. MATH-384, MATH-768, and GSM8K-384 use 20 examples, three repeats, and interleaved method order. Because timing estimates on a smaller code subset were unstable, MBPP-384 uses all 257 historically observed examples with three repeats. CUDA synchronization brackets each model call. End-to-end generation time includes generation-side orchestration and excludes dataset loading, task scoring, and cluster scheduling. Mini and Flash remain resident on `cuda:0` and `cuda:1` for every method, including single-model baselines, so placement is held fixed.

Experiments were conducted on two NVIDIA H200 GPUs, with approximately 139.8 GiB of runtime-visible memory per device. The main timing table reports the maximum reserved-memory measurement for this two-GPU deployment. The secondary one-repeat MBPP-512 timing diagnostic is reported in Appendix H.

Only runs passing completeness checks are included in the analysis. These checks verify that each required method and configuration has the expected number of evaluation outputs and timing measurements.

E Additional Quality–Latency Panels

Figure 3 presents the MATH-384 and MBPP+-384 Plus trade-offs, complementing the two representative settings shown in the main text. These panels use the same frozen configurations, quality estimates, confidence intervals, and standardized timing measurements as Figure 2.

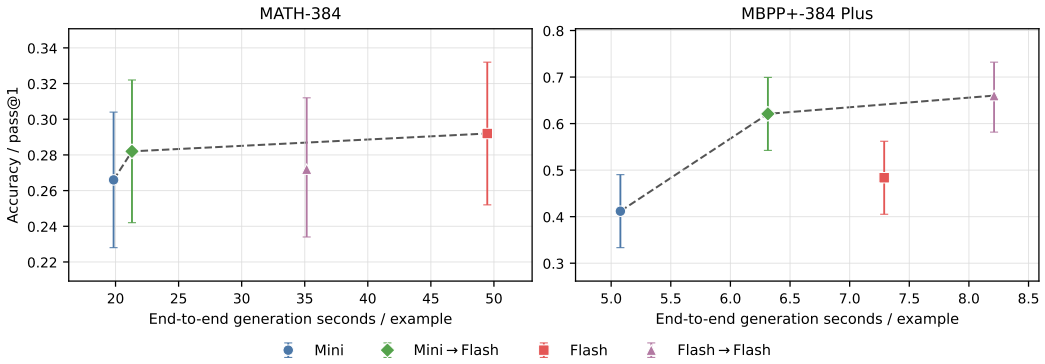

Figure 3: Additional quality–latency trade-offs for MATH-384 and MBPP+-384 Plus. Error bars show 95% bootstrap confidence intervals. Dashed lines connect nondominated methods.

F Termination and Length Details

F.1 MATH Length Sensitivity and Fixed-Span Failure Modes

The main failure mode appears when the draft does not contain enough of the response to repair. Table 5 reports termination behavior on the held-out MATH set. At 384 tokens, roughly three quarters of generations reach the cap. At 768 tokens the cap rate falls to 38–42%, but remains material.

Table 5: MATH quality and termination behavior at 384- and 768-token output caps.

Panel	Method	Score	Cap rate	EOS rate	Tokens p50 / p95
MATH-384	Mini	.266	.738	.262	384 / 384
	Flash	.292	.730	.270	384 / 384
	Mini→Flash	.282	.738	.262	384 / 384
	Flash→Flash	.272	.746	.254	384 / 384
MATH-768	Mini	.400	.422	.578	592.5 / 768
	Flash	.540	.378	.622	588.5 / 768
	Mini→Flash	.376	.418	.582	635.5 / 768
	Flash→Flash	.540	.396	.604	601.0 / 768

The longer budget benefits Flash far more than small-to-large correction. Flash reaches 0.540 at MATH-768, and same-model correction preserves that score at lower latency. Mini reaches 0.400, but Mini→Flash reaches only 0.376. The refiner can edit the span it receives but cannot add a missing continuation beyond it, and additional correction passes did not remove this gap during development. The strong MATH-384 trade-off therefore does not predict the ordering at longer budgets.

Table 6 separates outputs that reach the generation cap from those that terminate with EOS. The conditional accuracies are descriptive. Termination is associated with problem difficulty and model behavior, so their difference is not the causal effect of truncation.

Table 6: Descriptive MATH accuracy conditioned on reaching the output cap or terminating naturally.

Panel	Method	Cap rate	EOS rate	Capped acc.	Natural acc.
MATH-384	Mini	.738	.262	.092	.756
	Flash	.730	.270	.082	.859
	Mini→Flash	.738	.262	.089	.824
	Flash→Flash	.746	.254	.080	.835
MATH-768	Mini	.422	.578	.057	.651
	Flash	.378	.622	.053	.836
	Mini→Flash	.418	.582	.024	.629
	Flash→Flash	.396	.604	.091	.834

On the 95 project-untouched GSM8K problems, cap rates at 384 tokens are 10.5% for Mini, 3.2% for Flash, 10.5% for Mini→Flash, and 4.2% for Flash→Flash. At 512 they fall to 7.4%, 2.1%, 7.4%, and 0%, respectively. On the 153 project-untouched MBPP+-384 evaluation outputs, cap rates are 9.8%

for Mini and Mini→Flash, 0% for Flash, and 0.7% for Flash→Flash. These termination statistics describe the quality evaluation set, which is separate from the 257-task systems timing set. MBPP+ quality scores are obtained through the separate EvalPlus evaluation described in Appendix B.

F.2 MBPP+ 512-TOKEN SENSITIVITY

After the project-untouched MBPP+-384 results were known, we evaluated the same 153 tasks at 512 tokens. This tests length sensitivity on the same evaluation set and is not an independent confirmation. Mini, Flash, and Mini→Flash hold their configurations fixed apart from length. Flash→Flash changes its refinement threshold from 0.20 to 0.30 and is therefore a per-budget comparison rather than a pure length ablation.

Table 7: MBPP+ length sensitivity on the same 153 tasks. †Flash→Flash also changes its refinement threshold.

Method	384 Base / Plus	512 Base / Plus	Δ Base	Δ Plus
Mini	.510 / .412	.503 / .405	-.007	-.007
Flash	.575 / .484	.575 / .484	.000	.000
Mini→Flash	.745 / .621	.732 / .608	-.013	-.013
Flash→Flash†	.778 / .660	.784 / .660	+.007	.000

G MECHANISM DETAILS

The draft-only and four draft-conditioned correction arms use the same exact Mini draft token IDs for a problem; mask-only omits the draft. For each problem, the draft-conditioned refinement arms start from the same random-number-generator state. Local refinement edits independent 64-token generated chunks. Context-blockwise refinement keeps the full draft visible, pins positions outside the active block, and exposes earlier block updates to later blocks. Hybrid refinement applies context-preserving blocks before a full-global stage. Full-global refinement makes the entire generated span editable at once.

Full-global refinement makes 6.14 total edit events and 4.86 final token changes per GSM8K example, and 7.05 events and 5.73 final changes per MATH example. The difference reflects repeated edits to some positions: across 300 examples, GSM8K has 263 repeated edit events and MATH has 209. Full-global edits 1,578 distinct GSM8K positions and 1,906 MATH positions in total. It inserts EOS on 41 GSM8K and 11 MATH examples and never removes a pinned draft EOS.

Decoded text changes while the extracted answer remains fixed on 256/300 GSM8K and 271/300 MATH examples. These changes include formatting, wording, and reasoning text. Without manual semantic annotation, they should not be described as reasoning improvements. Among correctness transitions, defined as fixes plus harms, 13/15 are beneficial on GSM8K and 7/8 on MATH. These denominators exclude answer changes that leave correctness unchanged; the total numbers of extracted-answer changes are 20 and 9, respectively.

Mask-only uses the correction-stage budget rather than the total drafting-plus-correction budget. Under the native editing rule, it terminates after only 2.6 mean forward calls on GSM8K and 2.0 on MATH. Its failure shows that the small correction stage cannot recreate a response from masks; it is not a comparison against a fully budgeted all-mask decoder.

H SECONDARY SYSTEMS DIAGNOSTICS

The MBPP-512 timing panel uses all 257 historically observed tasks but only one measured repeat. It is therefore a secondary diagnostic. End-to-end times are 6.59 seconds for Mini, 7.76 for Flash, 7.95 for Mini→Flash, and 8.77 for Flash→Flash. Mini→Flash is 0.98× as fast as Flash and Flash→Flash is 0.89×. In this diagnostic, the Mini→Flash speed advantage observed at 384 tokens does not persist at 512 tokens. Accuracy on these historically observed tasks is excluded from the primary quality results.

I COMPARISON WITH EARLIER RESULTS

An earlier public implementation decoded and retokenized drafts before refinement and used different EOS and padding handling. All primary results use the corrected exact-token handoff and stopping

rules in Appendix A. Rerunning representative historical configurations with this implementation preserves the main qualitative trends. Table 8 compares the earlier results with corrected evaluations, distinguishing historical results from the locked and project-untouched results. Historical examples were observed during development, and the historical code results use the project's custom scorer rather than EvalPlus. They are excluded from the primary quality estimates.

Table 8: Earlier released results and corrected evaluations. Historical, locked, and project-untouched results are identified separately. M–F denotes Mini→Flash and F–F denotes Flash→Flash.

Setting	Released result	Corrected or new result
GSM8K-384 F–F	Flash .848, F–F .899, 1.20×	historical .848 to .900; project-untouched .789 to .916
GSM8K-512 F–F	Flash .850, F–F .907, 1.22×	historical .850 to .904; project-untouched .789 to .937
GSM8K-384 M–F	Mini / Flash / M–F .703 / .848 / .752	historical .703 / .848 / .752
MATH-384 M–F	.272 / .300 / .294, 2.17×	locked .266 / .292 / .282–.284; timed point 2.32×
MATH-512 M–F	.314 / .392 / .336, 1.92×	historical .336 / .402 / .362, about 2.01×
MBPP-384 M–F	.401 / .545 / .568, 1.11×	historical .401 / .545 / .588; project-untouched Base .510 / .575 / .745
MBPP-384 F–F	Flash .545, F–F .693	historical .545 to .735; project-untouched Base .575 to .778
HumanEval-384 F–F	Flash .415, F–F .622	historical .415 to .720; historically observed custom scorer

In the corrected historical runs, GSM8K-384 Flash→Flash produces 115 fixes and 56 harms relative to Flash; Mini→Flash produces 64 fixes and 10 harms relative to Mini. MBPP-384 Flash→Flash produces 66 fixes and 17 harms relative to Flash, while Mini→Flash produces 56 fixes and 8 harms relative to Mini. Counts use unique problem IDs, correcting the earlier treatment of repeated deterministic outputs as separate pairs.

Earlier Flash controls selected within latency windows are excluded from the primary comparisons because their development and evaluation sets overlap on some code benchmarks. HumanEval and synthetic results are also retained only as historical diagnostics.

J Reproducibility Details

Each reported run retains its fixed configuration, examples, model revisions, execution environment, and generation outputs. Raw outputs are preserved separately from derived results, and the analysis scripts are deterministic. The quality–latency figures combine quality and timing measurements from separate evaluations of the same configurations. The configurations, model revisions, and evaluation procedures are specified above. The codebase will be released publicly.