Speculative Decoding for Hybrid SSM–Attention Models: A Case Study with PLaMo 2 on llama.cpp

Gurunath Lunkupali Venugopal

June 2026

Abstract

Speculative decoding is a standard technique for accelerating autoregressive LLM inference, but it is poorly supported for hybrid state-space-model (SSM) architectures: the fixed-size recurrent state of Mamba-style layers cannot be truncated on token rejection the way a KV cache can. This was long a practical blocker—as recently as late 2025, vLLM rejected all speculative-decoding methods on Mamba-hybrid models outright—but by mid-2026 the rollback problem had been solved generically across several stacks (llama.cpp, and conditionally vLLM and SGLang), which independently converged on the same snapshot-and-restore primitive (Section 6). **We run our experiment on llama.cpp on a single laptop** (Apple M4 Pro, Metal backend, build b9596) and do not benchmark the other engines. We measure `pfnet/plamo-2-8b` (target) with `pfnet/plamo-2-1b` (draft) across three task families with greedy sampling, verifying byte-identical outputs versus the non-speculative baseline. The result is a carefully measured *negative* one: despite token acceptance rates up to 90% and a draft model that is 6.3× faster standalone, draft-model speculation is never profitable on this hardware (best case 0.99×; worst 0.29×). Only model-free n-gram (prompt-lookup) drafting yields a speedup, and only on translation (1.21×). We attribute the slowdown to per-round costs—serial draft decoding, batched verification, and ∼33.5 MiB SSM-state checkpoints—that exceed the savings on bandwidth-bound consumer hardware. Along the way we discovered a novel quantization bug: quantizing PLaMo 2's Mamba output projection (`ssm_out`) to Q8_0 causes degenerate generation on newline-containing prompts in both the 1B and 8B models; we isolate the faulty tensor experimentally and provide a one-flag mitigation that likely applies to all community PLaMo 2 GGUFs.

1 Introduction & Motivation

PLaMo 2 is Preferred Networks' (PFN) family of Japanese/English foundation models built on a Samba-style hybrid architecture that interleaves Mamba2 state-space (SSM) layers with sliding-window attention [1]. Hybrid SSM–attention models are attractive for inference because the SSM layers replace an ever-growing KV cache with a fixed-size recurrent state, but that same property breaks one of the most widely used inference accelerations: speculative decoding.

Speculative decoding drafts k candidate tokens cheaply (with a small draft model, or model-free heuristics such as prompt lookup), verifies them in a single batched forward pass of the target model, and—under greedy sampling—is *lossless*: output is provably identical to plain decoding. The catch for hybrid models is rejection. With a pure transformer, rejecting speculated tokens means truncating the KV cache, an $O(1)$ bookkeeping operation. With an SSM layer, the recurrent state after processing k tokens has irreversibly mixed those tokens in; there is no truncation operator. This was historically a practical blocker: as recently as late 2025, vLLM raised a `NotImplementedError` ("Mamba with speculative decoding is not supported yet") for *all* speculative methods on Mamba-hybrid models [2]. By the time of our measurements (mid-2026) the problem had been solved generically—first in llama.cpp via per-sequence rolling checkpoints of SSM state [3], and subsequently in vLLM and SGLang, which converged on the same primitive (we survey the full landscape in Section 6).

Our experiment uses llama.cpp on a local MacBook; we did not benchmark vLLM or SGLang. We chose it for a clean, accessible implementation on Apple Silicon, to ask an empirical question that, to our knowledge, has not been answered for PLaMo 2: *does speculative decoding actually pay off on a hybrid*

SSM–attention model, once correctness is solved? We answer it for the PLaMo 2 8B/1B pair, with a rigorous losslessness check, and report an honest negative result with a mechanistic explanation. As a secondary, first-class contribution, we document and root-cause a PLaMo 2 quantization bug that silently corrupts generation in standard Q8_0 GGUFs.

2 Background

2.1 Samba-style hybrid architecture

PLaMo 2 follows the Samba recipe: blocks of Mamba2 SSM layers interleaved with sliding-window attention layers [1]. The SSM layers carry a fixed-size recurrent state (convolution state plus SSM state) per sequence; the attention layers keep a bounded sliding-window KV cache. This gives near-constant memory during decode, at the cost of state that evolves destructively token by token.

2.2 KV truncation vs. SSM state rollback

In a transformer, speculative verification that accepts $a < k$ drafted tokens simply discards KV entries for positions beyond a. An SSM has no analogous operation: its state is a lossy summary of the whole prefix, so "forgetting" the last $k - a$ tokens requires having saved an earlier copy of the state. This is exactly why vLLM's speculative-decoding path originally refused Mamba-hybrid models for every proposer type (n-gram, draft model, EAGLE, Medusa, MTP) [2].

2.3 llama.cpp's recurrent checkpoint mechanism

llama.cpp addressed this generically in its recurrent memory module: `llama_memory_recurrent` maintains a rolling buffer of checkpoints (depth 8 per sequence) of the SSM state tensors, saved at batch boundaries and restored when speculation is rejected [3]. Correctness becomes architecture-agnostic, but the checkpoints are not free: for the PLaMo 2 8B, server logs in our runs reported approximately 33.5 MiB per context checkpoint. Every speculation round therefore adds state-copy traffic on top of draft decoding and batched verification—a cost model that turns out to matter a great deal in Section 6.

3 A Quantization Bug Found Along the Way

Before any speculation numbers could be trusted, we hit a correctness bug that we believe affects all community PLaMo 2 GGUFs and present as a first-class contribution.

Symptom. In the first benchmark run, every prompt containing a bare newline (token 10, the reserved byte token `0x0A`)—i.e. all code and translation prompts—returned *empty* content while "generating" 256 tokens at full speed. This occurred with both the 1B and the 8B Q8_0 models.

Root-cause chain. Each step below is one controlled experiment:

1. Requesting raw token IDs (`return_tokens: true`) revealed that the model emits token 31, i.e. `<|plamo:reserved:0x1F|>`, indefinitely; reserved tokens render as empty text and are not end-of-generation, so decoding runs to the limit.
2. Tokenizer round-trip via `/tokenize` + `/detokenize` is perfect, including indented code. Not a tokenizer bug.
3. Full prompt token-ID comparison, HF transformers vs. llama.cpp: identical (modulo BOS; manually prepending BOS changed nothing).

4. Ground truth via HF transformers on CPU in fp32 (this required pinning `transformers<4.58` *and* writing pure-PyTorch shims for the `causal_conv1d` and `mamba_ssm` reference functions, since pfnet's modeling code's CPU fallback still calls those packages): the reference implementation generates correct code for the failing prompt. Hence a llama.cpp-side problem.
5. Unquantized bf16 GGUF in llama.cpp: correct output. Hence a quantization problem, not an inference-kernel problem.
6. Re-quantizing to Q8_0 from bf16 with `llama-quantize`: still broken. Hence not a converter bug.
7. Per-tensor isolation with `--tensor-type X=bf16` (Table 1): keeping `ssm_out` alone in bf16 fixes generation; exempting any other tensor does not.

Table 1: Per-tensor isolation: which single tensor kept in bf16 (all else Q8_0) repairs generation on newline-containing prompts.

Tensor(s) kept in bf16	Role	Output
token embeddings	embedding	broken
output	LM head	broken
all `ssm_*` matrices	all Mamba projections	**fixed**
`ssm_in`	Mamba input projection	broken
`ssm_x`	Mamba x projection	broken
`ssm_dt`	Mamba Δt projection	broken
`ssm_out`	Mamba output projection	**fixed**

Conclusion and mitigation. Quantizing PLaMo 2's Mamba output projection (`ssm_out`) to Q8_0 causes degenerate generation on newline-containing prompts. The mitigation is one flag:

```
llama-quantize --tensor-type ssm_out=bf16 model-BF16.gguf model-Q8fixed.gguf Q8_0
```

Impact. The bug reproduces on both the 1B and 8B models and is a property of the standard Q8_0 recipe, so it likely affects all community PLaMo 2 GGUFs published to date. Reported upstream as ggml-org/llama.cpp#24501; a natural fix is to protect `ssm_out` in llama.cpp's plamo2 quantization recipe. All benchmarks in this paper use the repaired quantization ("Q8fixed": Q8_0 with `ssm_out` in bf16) for *both* models.

4 Methodology

Hardware and build. MacBook Pro with Apple M4 Pro, 48 GB unified memory, Metal backend. llama.cpp build b9596 (official release binaries), served via `llama-server`.

Models and quantization. Target: `pfnet/plamo-2-8b`. Draft: `pfnet/plamo-2-1b`. Both converted to GGUF and quantized to Q8_0 with `ssm_out` kept in bf16 (Section 3). Standalone, the draft decodes at 134 tok/s vs. 21.4 tok/s for the target—a 6.3× ratio that would normally make draft-model speculation attractive.

Configurations.

- `fx_baseline`: plain decoding, no speculation.
- `fx_spec_d{2,4,8,16}`: draft-model speculation (`--spec-type draft-simple`) with `draft-n-max` = $2, 4, 8, 16$ and `draft-n-min` = 1.
- `fx_ngram`: model-free n-gram speculation (`--spec-type ngram-simple`, prompt lookup; no draft model).

Tasks and protocol. Twelve prompts spanning three task families —`ja_chat`, `translation_en_ja`, and `code`—each run 3 times per configuration (`bench/prompts.json`). Greedy sampling (temperature 0, top-k 1), `n_predict` 256. We record decode throughput, drafted/accepted token counts, and a SHA-256 hash of the full generated text per prompt.

Losslessness verification. For every prompt, the SHA-256 of the speculative output is compared against the baseline output. All speculative configurations produced byte-identical greedy output to the baseline, confirming that llama.cpp's SSM checkpoint/rollback is exact.

5 Results

Table 2 reports median decode throughput per task, speedup versus baseline, and token acceptance rate. The headline: with a real draft model, speculation never wins on this hardware. The best draft-model cell is `fx_spec_d2` on translation at 0.99× (break-even); ja_chat degrades to 0.29× at depth 16. The only profitable configuration anywhere is n-gram drafting on translation (1.21×), with `fx_spec_d4` on translation near-breakeven-positive at 1.18×.

Table 2: Median decode throughput in tok/s (speedup vs. baseline) [token acceptance rate]. Greedy sampling; all speculative outputs byte-identical to baseline.

Config	code	ja_chat	translation_en_ja
`fx_baseline`	19.7 (1.00×)	21.6 (1.00×)	15.7 (1.00×)
`fx_ngram`	18.1 (0.92×) [19%]	17.6 (0.81×) [63%]	**19.0 (1.21×) [23%]**
`fx_spec_d2`	16.5 (0.84×) [90%]	13.8 (0.64×) [80%]	15.5 (0.99×) [87%]
`fx_spec_d4`	12.0 (0.61×) [77%]	10.2 (0.47×) [55%]	18.5 (1.18×) [70%]
`fx_spec_d8`	11.6 (0.59×) [55%]	7.9 (0.36×) [32%]	11.1 (0.71×) [48%]
`fx_spec_d16`	11.7 (0.59×) [33%]	6.2 (0.29×) [17%]	9.3 (0.59×) [29%]

Figure 1 summarizes speedups per task and configuration; Figure 2 shows acceptance rates, which decay sharply with draft depth; Figure 3 shows full throughput distributions across prompts and repeats; Figure 4 relates acceptance to realized speed, making the central paradox visible: 90% acceptance still loses.

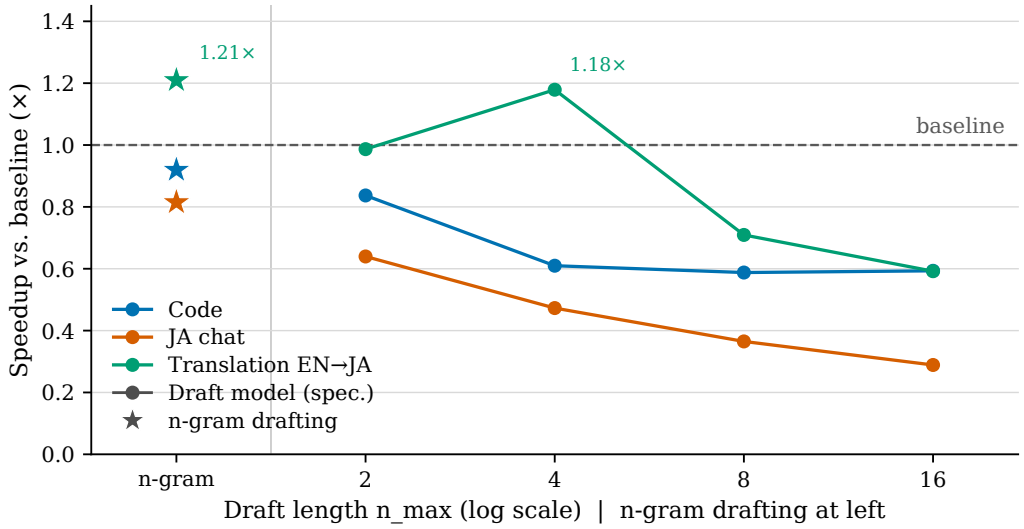

Figure 1: Decode speedup vs. baseline by task and configuration. Only n-gram drafting on translation exceeds 1.0×; draft-model speculation is unprofitable everywhere.

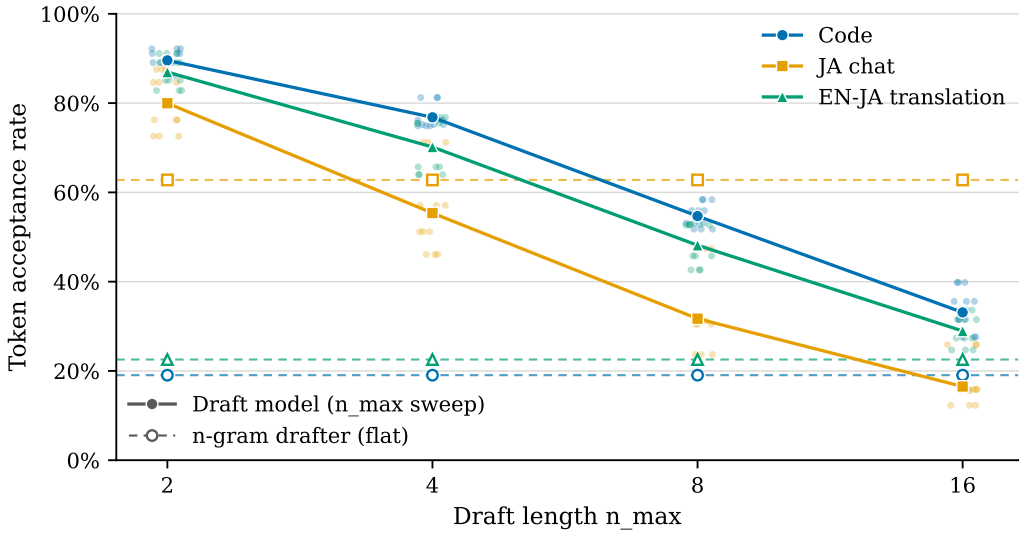

Figure 2: Token acceptance rate by task and draft depth. Acceptance is excellent at depth 2 (up to 90%) but per-token acceptance decays with depth, so deeper drafts amortize worse.

Losslessness. All speculative configurations produced byte-identical greedy output to baseline, verified by SHA-256 per prompt. The negative result is therefore purely about throughput, not correctness.

6 Discussion

Why 90% acceptance still loses. Each speculation round pays three costs that plain decoding does not: (i) *serial draft-model decode*—the 1B draft, though 6.3× faster standalone, still runs serially before each verification and competes for the same unified-memory bandwidth; (ii) *batched verification* in the target, which on bandwidth-bound Metal hardware is far from free relative to single-token decode (single-token decode already saturates memory bandwidth, so a k-token verify costs nearly as much per token as plain decode); and (iii) *SSM-state checkpointing*—roughly 33.5 MiB saved per context checkpoint for the 8B, plus restores on rejection. At depth 2 with 90% acceptance (code), these round costs already eat the entire benefit (0.84×). Deeper drafts amortize per-round costs over more tokens in principle, but per-token acceptance decays with depth (Figure 2), so realized acceptance lengths grow slowly while rejected work and rollbacks grow quickly: depth 16 lands at 0.29–0.59×.

Why n-gram wins on translation. Prompt-lookup drafting has essentially zero proposal cost—no draft forward pass, no second model resident in memory. Translation outputs copy long spans from the source prompt verbatim (names, numbers, code-switched terms), so even a modest 23% acceptance converts into free accepted tokens, yielding the study's only speedup (1.21×). On code, ja_chat, and any task where output text is not substring-recoverable from the prompt, lookup proposals miss and the verification overhead makes it a net loss.

Implications. These numbers temper the urgency that was, at the time of writing, attached to lifting vLLM's Mamba-hybrid speculative-decoding restriction [2]: correctness machinery (as llama.cpp shows) is solvable, but a separate draft model may simply not pay on bandwidth-bound hardware, and the checkpoint traffic is an architecture-intrinsic tax. Cheap proposers (n-gram, and especially self-drafting methods such as MTP or EAGLE-style heads that reuse the target's hidden states and add no second model) are the more promising direction for hybrids, since they attack the dominant cost—the serial draft—directly. Notably, PLaMo 3 NICT models move to an attention-only architecture, which sidesteps SSM rollback entirely and restores cheap KV truncation; speculative decoding economics there should look conventional again.

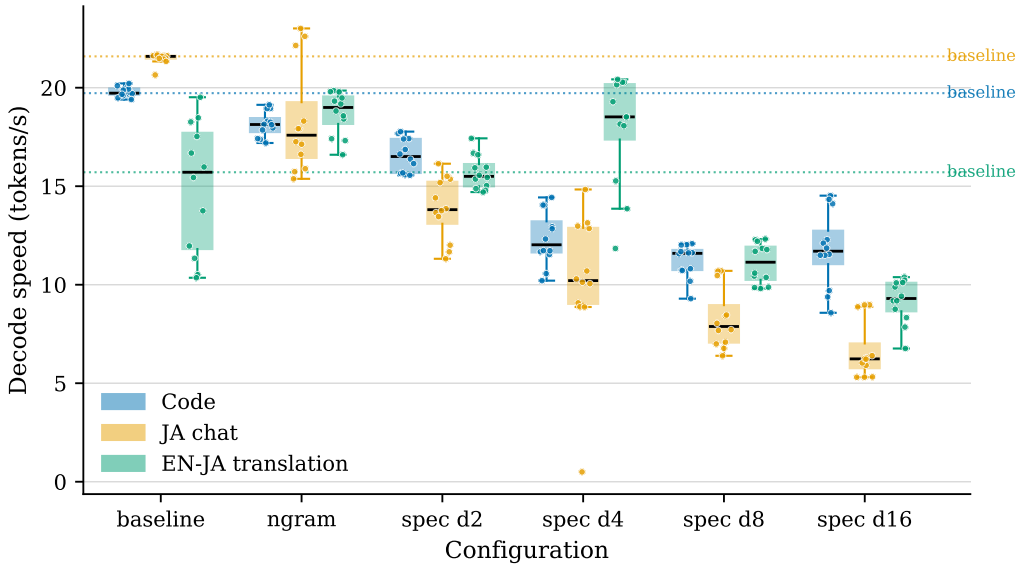

Figure 3: Decode throughput distributions across prompts and repeats for each configuration and task.

Framing. We deliberately report this as a measured negative result with a mechanism. "Speculation is lossless and works, but is unprofitable for PLaMo 2 8B/1B on M4 Pro Metal" is directly actionable for anyone deploying these models locally: run the baseline, or n-gram for translation-like workloads.

Platform landscape and scope. We benchmarked llama.cpp on a single laptop and did not test other engines—a deliberate scope choice, not a claim that llama.cpp is unique. As of our measurements (mid-2026), the SSM-rollback problem had been solved in at least three stacks, which independently converged on the same snapshot-and-restore primitive:

- **llama.cpp** merged it as PR #19493 [3] (an earlier alternative, #20075, was closed unmerged): the `llama_memory_recurrent` rolling checkpoint buffer used here.
- **vLLM**, which into early 2026 rejected the combination outright (#30114 [2], closed April 2026), added it in PR #33726 [4] (merged Feb 2026) and now supports it *conditionally*—with `--mamba-cache-mode align` plus self-drafting heads such as EAGLE3 or MTP—though model-free n-gram drafting can still corrupt recurrent state on some hybrids.
- **SGLang** added it in PR #13434 [5] (merged Dec 2025) and ships the same primitive on NVIDIA hardware: one isolated recurrent-state slot per draft token, plus a convolution-window snapshot/restore.

The open question across all three has shifted from *whether* rollback is correct to *which* proposer types preserve state exactly—precisely the property our SHA-256 byte-identical check measures. Because our negative result concerns the *economics* of a separate draft model on bandwidth-bound hardware—not the correctness machinery—we expect it to be robust on similar hardware; but we have not measured vLLM or SGLang, and server-class GPUs with cheaper batched verification could plausibly change the verdict (see Limitations).

7 Limitations

- **Single hardware point.** All measurements are on one M4 Pro with Metal. CUDA GPUs with higher compute-to-bandwidth ratios and cheaper batched verification could flip the economics; we have no CUDA leg yet.

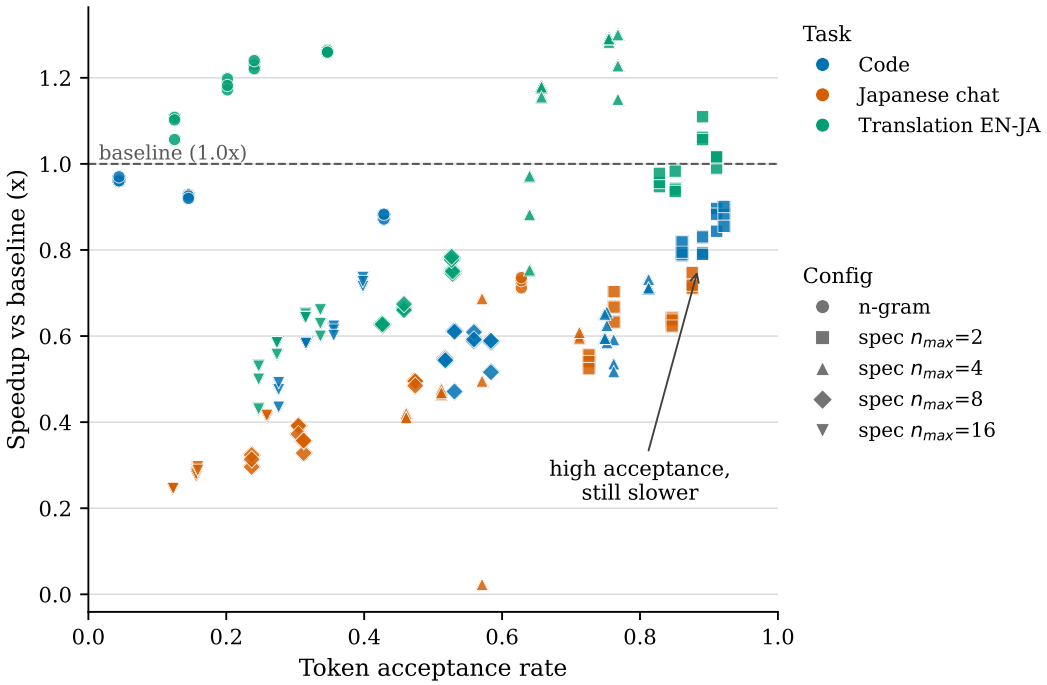

Figure 4: Acceptance rate vs. realized decode speed. High acceptance does not translate into speedup: per-round overheads dominate.

- **Small prompt suite.** 12 prompts × 3 repeats across three tasks captures task-level trends but not workload diversity (long-context, RAG, agentic loops with heavy prompt reuse—likely favorable to n-gram drafting).
- **One model pair and one quantization.** Results are for PLaMo 2 8B/1B at Q8fixed; other draft sizes, quantizations, or speculation parameters (e.g. adaptive depth) were not swept.
- **One engine.** We measured llama.cpp only. vLLM and SGLang now implement the same hybrid speculative-decoding rollback (see Section 6) but were not benchmarked; their batching and checkpoint costs differ, so the throughput verdict may not transfer to them or to server-class GPUs.

8 Reproducibility

All artifacts live under `/Users/gurunathlunkupalivenugopal/ionet/pfn-try`:

- `bench/prompts.json` — 12 prompts across the three tasks.
- `results/fx_baseline.jsonl`, `results/fx_spec_d{2,4,8,16}.jsonl`, and `results/fx_ngram.jsonl` — one JSON object per line with fields `label`, `task`, `prompt_idx`, `rep`, `tok_per_sec`, `n_predicted`, `draft_n`, `draft_n_accepted`, `prompt_per_sec`, `content_sha`, `content`.
- `report/figures/` — figure PDFs plus the Python scripts that generate them.
- `commands-log.md` — the full command log, including the `ssm_out` bug-hunt chain (section 3) and harness fixes (section 4).

Key commands:

```
llama-quantize --tensor-type ssm_out=bf16 plamo-2-8b-BF16.gguf plamo-2-8b-Q8fixed.gguf Q8_0
llama-server -m plamo-2-8b-Q8fixed.gguf -md plamo-2-1b-Q8fixed.gguf --spec-type draft-simple
--draft-max {2|4|8|16} --draft-min 1
llama-server -m plamo-2-8b-Q8fixed.gguf --spec-type ngram-simple
```

Two harness pitfalls worth repeating: on build b9596, `-md` alone loads the draft model but leaves speculation *off* (the log prints "no implementations specified for speculative decoding"); `--spec-type draft-simple` is required, and engagement should be confirmed via the `timings.draft_n` / `draft_n_accepted` fields in responses. Greedy losslessness should be checked with a real content hash (SHA-256), not Python's per-process-salted `hash()`.

References

[1] Preferred Networks. *PLaMo 2 Technical Report*. arXiv:2509.04897, 2025. https://arxiv.org/abs/2509.04897

[2] vLLM project. "Speculative decoding support for mamba models." GitHub issue #30114, vllm-project/vllm (opened Dec. 2025, closed Apr. 2026). https://github.com/vllm-project/vllm/issues/30114

[3] ggml-org/llama.cpp. "server : speculative checkpointing" — recurrent-state checkpoint/restore for speculative decoding on SSM/recurrent models. Pull request #19493 (merged Apr. 2026); supersedes the closed alternative #20075. https://github.com/ggml-org/llama.cpp/pull/19493

[4] vllm-project/vllm. "[Model][Spec Decode] Nemotron-H MTP and Mamba Speculative Decoding Support." Pull request #33726 (merged Feb. 2026). https://github.com/vllm-project/vllm/pull/33726

[5] sgl-project/sglang. "[Spec] Mamba2 support in target models." Pull request #13434 (merged Dec. 2025). https://github.com/sgl-project/sglang/pull/13434