PURETENSOR
INTELLIGENCE DISTILLED

When Grammar Meets Speculation

A constrained-decoding deadlock under overlapped speculative decoding, and the flag that closed it

Paper: PT-R-2026-029 v1.0

Author: PureTensor Ltd

Date: 28 September 2026

Status: Published

Abstract

We serve DeepSeek-V4.1-Flash through SGLang as an always-on, OpenAI-compatible endpoint for our own agents. The model is split four ways across two compute nodes, with speculative decoding (DSpark) and SGLang's default overlap scheduler both enabled. At 05:41:46 UTC on 27 September 2026 the server logged its last batch and went silent. A grammar-constrained request (a JSON-schema response format) had just started decoding alongside an unconstrained agent loop. The scheduler watchdog killed the process 300 s after the stall began, and the first outage lasted about 12 minutes.

In a harness run against the live server, a constrained request running concurrently with the loop hung the server in 2 of 2 attempts, or 3 of 3 counting production. The hang arrived 24 to 47 s after the constrained request was submitted. The same prompt without the response format ran clean at the same concurrency. The constrained request ran clean on its own in 2 of 2 runs, although only a summary records those runs. At the hang all four GPUs read 100% utilisation while drawing 89.7 to 104.4 W against a 250 W cap. Every scheduler rank we captured was parked on the same host stack, waiting on the device copy that the grammar mask depends on.

The obvious remedy, disabling overlap scheduling, failed on first decode with a TypeError. That left the service in a crash loop of 22 failed starts, and it went unnoticed for about 1 h 41 min because our verification step polled the health endpoint in a silent, unbounded loop. Counting the revert's boot, the endpoint was unavailable for about 1 h 46 min. A five-line patch made the non-overlap path run, at a throughput cost of 6% single-stream and 4% at concurrency 4. The deadlock survived: 1 of 3 patched replicas hung, and the other two lost concurrency partway through. The host wait had moved to a different frame, and the device still stalled.

The fix was one flag, `--grammar-backend none`, which makes SGLang refuse constrained requests at admission with an HTTP 400 in 0.17 to 0.18 s. Over one server invocation with the hang monitor armed, 100 of 100 constrained arrivals were rejected and 454 concurrent agent turns completed, with 0 hangs. Throughput read 91.9 / 242.6 tok/s at concurrency 1 / 4, against a baseline of 91.9 / 235.5. Constraint enforcement moved to the clients. One client fell back silently to a hosted model for part of the rollout window.

These counts are small and come from one build on one topology. The device-side root cause was not established. The claim that survives is that on this deployment a grammar-masked DSpark verify sharing a batch with another request deadlocks the server, and that refusing such

requests at admission removes the trigger at no measurable cost.

1. The Deployment

The endpoint runs DeepSeek-V4.1-Flash on SGLang, built from the DeepSeek-V4.1 branch; the image reports its version as 0.0.0.dev0. We shard it with tensor parallelism 4 and expert parallelism 4 (`--tp 4 --ep-size 4`). Tensor parallelism splits each layer's matrices across devices, and expert parallelism places different mixture-of-experts experts on different devices. The four GPUs sit in two compute nodes, two RTX PRO 6000 Blackwell-class cards per node, each power-capped at 250 W, joined by a 200 Gb/s RoCE fabric. One consequence governs everything that follows. A single forward pass is a sequence of collectives, operations that all four ranks (one process per GPU) must enter together. If one rank never arrives, the others wait.

The relevant server settings, all public SGLang flags, are a context length of 262,144 tokens, at most 8 running requests, chunked prefill of 2,048 tokens, and a scheduler watchdog of 300 s. When the watchdog expires, the process tree is killed and the service manager restarts it. The model's Engram memory tables are held in host RAM.

Three features interact in this incident.

Speculative decoding. We use DSpark, the draft-and-verify method SGLang ships for DeepSeek-V4.1 (`--speculative-algorithm DSPARK --speculative-dspark-block-size 5`). A cheap drafter proposes a block of up to 5 tokens. The target model verifies the whole block in one pass and keeps the accepted prefix. Just before the first hang, acceptance was 3.35 to 4.29 tokens per step, with an accept rate of 0.47 to 0.66.

Overlap scheduling. This is SGLang's default ("spec-v2 overlap"). The scheduler prepares batch N+1 on the CPU while the GPU is still running batch N, so the host rarely waits on the device.

Grammar-constrained decoding. When a request carries `response_format` json_schema (or regex, EBNF, structural tag), or `tool_choice: "required"`, SGLang compiles a grammar, using xgrammar by default. At every decode step it masks the vocabulary down to the tokens the grammar allows. Speculation complicates this. A mask must be built for each drafted position, and building it needs the grammar state after the previous step's accepted tokens. The host therefore has to wait for device results before it can proceed. That wait is a host-side barrier.

The endpoint became always-on on 26 September. Five kinds of client were using it on 27 September:

- an observer agent loop with roughly 86k tokens of context, `tool_choice` auto, sending one short turn every one to three seconds;
- a triage agent in a shadow trial, which sent one json_schema-constrained request of roughly 33k tokens with max_tokens 64,000;
- an hourly conformance probe using `tool_choice=required`;
- a memory-extraction client sending `response_format` json_object;
- an internal gateway passing `response_format` and `tool_choice` through from its own clients.

The server invocation that hung had started at 17:24:07 UTC on 26 September, about 12 h 18 min before the hang. Internal notes describe about 12.5 h of clean running with hundreds of 2- to 8-way concurrent batches, all unconstrained. The "hundreds" figure comes from a summary and was not re-counted. The journal independently confirms about 12.3 h of that invocation without a watchdog fire.

2. The Incident

The service journal and our reproduction logs are both in UTC, and we use UTC throughout. Some internal summaries labelled the first hang "05:41 BST". The journal shows the last logged batch at 05:41:46 UTC, and we go with the journal.

Time (27 Sep, UTC)	Event
05:41:46	Last batch logged; the constrained triage request is decoding alongside the observer loop (#running-req 2)
05:48:07	Scheduler watchdog (300 s) fires on the rank-0 node's two ranks
05:49:54 / 05:54:00	Service restarted / serving again

Nothing in the logs between the last batch and the watchdog fire indicated a problem, and no error was logged. The first outage lasted about 12 minutes. The triage request was the first constrained request this server invocation had seen alongside other traffic.

3. Reproduction

We wrote a driver that ran four experiments against the live server:

- **C**, the incident replica: an observer-like loop plus the constrained, roughly 33k-token triage request with max_tokens 64,000.
- **A**, the control: the same, with `response_format` removed.
- **B**: the constrained request alone.
- **S**: the loop plus a short constrained request every ~5 s.

The loop starts at about 80k to 118k tokens of context and grows by about 1.6k to 1.8k tokens per turn, with max_tokens 256 per turn and `tool_choice` auto.

We also ran a hang monitor on the rank-0 node. It declared a stall when no Prefill or Decode batch had been logged for 45 s while the server still reported running requests. It then captured GPU state (nvidia-smi) and Python stacks (py-spy, a sampling profiler that can dump a live process's stack) for every process on both nodes. This happens well before the 300 s watchdog. The monitor exists because of an earlier hang on 10 September, described in Section 4, whose stacks were lost: capture had run after the watchdog had already killed the processes.

Run	Condition	Result	Detail
Production, 05:41	constrained triage + observer loop	HANG	about 12 min outage
C1, 06:15	constrained + concurrent loop	HANG	scan sent 06:15:29; last good turn 06:16:16; 20 good loop turns; turn 21 never returned
C2, 06:27	constrained + concurrent loop	HANG	scan sent 06:28:05; last good turn 06:28:29; 8 good loop turns
A, 06:41	same prompt, unconstrained, concurrent	clean	72 loop turns, context up to 220,914 tokens; scan finished in 170.48 s, 16,257 completion tokens (13,495 reasoning), finish "stop"
B (x2)	constrained, alone	clean 2/2	from internal summary; driver logs not retained

Run	Condition	Result	Detail
Overnight	unconstrained concurrent traffic	clean	about 12.3 h of the hung invocation

The last good loop turn came about 47 s after the constrained scan was sent in C1 and about 24 s after it in C2. Internal summaries describe the hang as arriving "within 1 to 3 min of overlap". The logs support the shorter figure. Each harness hang cost a full watchdog cycle and reboot: C1 was captured at 06:17:03, the watchdog fired at 06:21:29, and the server was serving again at 06:27:15. C2 was captured at 06:29:16, the watchdog fired at 06:34:44, and the server was serving again at 06:40:41.

The controls separate the ingredients. Concurrency without a grammar is clean, both in control A and in the overnight traffic. A grammar without concurrency is clean in B, on summary evidence only. Speculation and overlap were on in every row. On this build, then, the trigger is a grammar-constrained request decoding in the same batch as another request. Control A also shows that the scan itself is not pathological. Unconstrained, it ran for 170.48 s to a natural stop while the loop's context grew past 220k tokens.

4. The State at the Hang

The two harness captures, at 06:17:03 and 06:29:16, agree.

Capture	GPU utilisation	Power per GPU (W)
1 (06:17:03)	100% on all four	104.43, 92.06, 97.22, 99.88
2 (06:29:16)	100% on all four	102.05, 89.69, 95.90, 97.71

A GPU doing real work at 100% utilisation on this card draws far more than 90 to 105 W. Utilisation here means only that a kernel is resident. The pattern of full utilisation at well under half the power cap is what kernels spinning in a collective that never completes look like. If we had been watching the utilisation graph, the service would have looked busy.

On the host, every scheduler rank we captured shows the same MainThread stack. It starts in the overlap event loop and passes through `run_batch`, the DSpark v2 worker's `forward_batch_generation` and `_forward_decode`, then `build_grammar_vocab_mask`, `_advance_pending_grammar` and `advance_grammar_fsm`. It ends at `result.copy_done.synchronize()`, blocked in a CUDA stream synchronise. The coverage is uneven. Capture 2 has this stack on all four ranks (TP0 to TP3). Capture 1 has it on TP0 and TP1, but its second-node dump did not include the scheduler processes. "Identical on all four ranks" is therefore directly evidenced by one capture, and by two ranks in the other.

Our reading is as follows. No rank diverged on the host. All four are waiting for a device-side copy that sits in the stream behind a cross-rank collective, and that collective never completes. The grammar barrier runs after the target-verify launch, so the host is waiting on a stream that already holds the verify's collectives. We did not establish the exact device-side cause. One hypothesis is that ranks disagree on accepted draft lengths under the grammar mask, which would give mismatched collective sizes. It remains a hypothesis.

The 10 September hang is worth setting beside this one, because from the outside the two looked the same: the server stops and the watchdog fires. That hang involved the same model and the same DSpark method, in a different deployment mode. A load loop (8 concurrent requests with 8,192-token inputs) hung at case 26 after about 10 minutes. NCCL debug logs showed that all four ranks had enqueued an identical collective sequence, 70,682 operations each with identical tails. GPU utilisation, however, was asymmetric. One node's GPUs were at 100% (spinning) and the other node's were at 0%, because their streams never reached the kernel. The cause was a host callback inside the decode stream that read the Engram tables

from NVMe with direct I/O and stalled under DSpark's variable gather sizes. Moving the tables into host RAM fixed it: 92 cases over 30 min, more than 513 requests, 0 hangs. In NVMe mode, by contrast, there had been a hang at case 26 after about 10 min that evening and 3 hangs in about 75 min earlier the same day.

The two hangs share a symptom and nothing else we can see. 10 September showed asymmetric GPUs and a stalled host callback. 27 September showed a symmetric 100% spin on all four GPUs, with every rank parked at the same grammar barrier. The lesson we took from 10 September, capturing stacks before the watchdog fires, is the reason we have the 27 September stacks at all.

5. Attempt 1: Disable Overlap Scheduling

The stacks pointed at the overlap path's grammar barrier, so we hypothesised that running without overlap would avoid it. At 07:24:06 we added `--disable-overlap-schedule` on both nodes and restarted.

Every start died on the first decode with `TypeError: DSparkWorkerV2.forward_batch_generation() got an unexpected keyword argument 'pp_proxy_tensors'`. SGLang's non-overlap speculative branch passes `pp_proxy_tensors`, and the DSpark v2 worker's signature does not accept it. As far as we can tell, this combination had never run.

The journal records 22 failed starts between 07:24:31 and 09:05:47; the service manager's restart counter reached 22 at 09:05:42. Each cycle took about 4 min to boot. The process died about 10 s after the server reported itself started, and restarted after a delay of about 31 s. Internal notes say "21 times". The journal shows 22 exits and 22 automatic restarts, the 22nd of which the revert stopped 4 s later. We use 22.

The crash loop lasted about 1 h 41 min (07:24:31 to 09:05:47). The endpoint was without service for about 1 h 46 min, from the 07:24:06 stop to 09:10:14, when the reverted configuration was serving again on both nodes. The observer loop depends on this endpoint and was down for the whole period.

No alert fired because the verification step waited for the health endpoint in an unbounded, silent loop. It polled every 10 s until it received HTTP 200 and printed nothing while it waited. The server never became healthy, so the loop neither succeeded nor reported. It was noticed at about 09:05. The failure itself was visible within one boot cycle of about 4 minutes. Our procedure was what hid it.

6. Attempt 2: Patch the Non-Overlap Path

We wrote a five-line patch that adds `pp_proxy_tensors=None` to the DSpark v2 worker's `forward_batch_generation`, mounted it over the image's file on both nodes, and kept `--disable-overlap-schedule`. Its unit tests passed 4/4.

The first guarded rollout, at 09:56, reverted itself wrongly and sent a false critical page. Its exception check read a journal window that began before the restart, and it matched the old process's shutdown traceback (a CancelledError). We scoped post-restart checks to the new process invocation, using the service manager's InvocationID, and rolled out again. The second guarded rollout restarted at 09:58:14. It produced its first real completion 260 s later, confirmed that overlap was disabled on both nodes, and was stable for 60 s.

Running without overlap has a throughput cost.

Throughput (1,024 forced output tokens, median of 3)	c1 tok/s	c4 tok/s
Baseline (overlap on, grammar on)	91.9	235.5
Patched non-overlap	86.1 (minus 6%)	226.4 (minus 4%)

We then replayed the incident.

Replica	Overlap window	Result
P1, 10:05:47	loop hit the 256k context limit at turn 85 (10:09:45), then HTTP 400 on every turn; about 223 s of real overlap in a 436.56 s scan	no hang; scan completed 64,000 tokens (finish "length")
P2, 10:13:21	loop hit the context limit at turn 85 (10:17:16); about 220 s of overlap in a 243.99 s scan	no hang; scan completed 23,045 tokens (finish "stop")
P3, 10:17:42	harness fixed to trim the loop below 200k; scan sent 10:17:57; last good loop turn 49 at 10:20:05	HANG; capture 10:20:54; watchdog 10:27:23

P1 and P2 are weaker evidence than they first appear. In both, the loop outgrew the context limit, so the constrained scan spent part of its life decoding alone. We fixed the harness to trim the loop below 200k so that concurrency could not lapse, and on the next run, P3, the server hung.

At the P3 hang all four GPUs were again at 100% utilisation, drawing 93.8 to 109.2 W. All four ranks' MainThread was in `run_batch` under the normal (non-overlap) event loop, at the host copy of `new_seq_lens`. The host frame was different and the device-side stall was the same. Without overlap the host waits somewhere else, but the device still stalls. On this evidence the deadlock is device-side whenever a grammar-masked DSpark verify shares a batch, whatever the scheduler mode.

The internal summary records this as "incident replica 3/3 hung". The logs show 1 hang in 3 patched replicas, and the first two did not hold concurrency for the whole scan. It is correct to say the deadlock survived the patch. It is not correct to say it reproduced 3 out of 3 times on the patched server. We reverted.

7. The Fix: Refuse Grammar at Admission

We went back to the original configuration and added one flag, `--grammar-backend none`. Overlap scheduling and DSpark were left as they were, and the patch files were removed. The rollout restarted at 10:32:10 and produced its first real completion 271 s later. The flag was confirmed on rank 0 and in both second-node scheduler processes, and the service was stable for 60 s.

With no grammar engine, SGLang rejects json_schema, regex, EBNF and structural-tag requests at admission, the point where a request is accepted into the scheduler. This includes `tool_choice=required`, and in practice json_object too. Rejected requests receive HTTP 400 with the message "Grammar-based generation (json_schema, regex, ebnf, structural_tag) is not supported when the server is launched with --grammar-backend none". No request ever reaches a decode step with a mask. The first such rejection in production was logged at 10:38:23.

We verified the fix on a single server invocation with the hang monitor armed.

Check	Result
Single json_schema request	HTTP 400 in 0.18 s

Check	Result
Single tool_choice=required request	HTTP 400 in 0.17 s
S test, 10:39:50 to 10:50:14	100/100 constrained rejected; latency min 0.17 s, median 0.21 s, max 16.62 s, 7 of 100 above 1 s; 248 loop turns, 0 loop errors, context up to 201,046 tokens
Two incident replicas	scans rejected in 3.52 s and 0.46 s; 0 hangs; overlap only about 20 s and 17 s
Control A, 10:50:53 to 11:00:00	64,000-token unconstrained scan completed in 529.73 s alongside 206 agent turns; 0 loop errors
Captures, watchdog fires, restarts, 10:38 to 11:00	0, 0, 0
Throughput c1 / c4 (tok/s)	91.9 / 242.6 (baseline 91.9 / 235.5)

Most of the evidence comes from the S test, which sent 100 constrained arrivals over 10.4 minutes against 248 concurrent turns, and from control A, which is the triage agent's production path now that it sends unconstrained requests. Across the two, 454 concurrent agent turns completed with 0 hangs. The two incident replicas mostly show that the request which wedged the server is now refused in under 4 s. Each run ended at the rejection, so the overlap they exercised was brief.

We did not investigate the 7 rejections that took longer than 1 s. They may have been queued behind the concurrent loop's long-context prefill, but that is a hypothesis only. The c4 figure rose by 7.1 tok/s (3%), which is within what three repetitions can resolve, so we read it as no measurable cost and do not claim a gain.

8. Moving the Constraint to the Clients

Refusing grammar at the server means each client has to enforce its own structure. The triage agent now sends unconstrained requests and validates its JSON against the schema itself. The hourly conformance probe skips its forced-tool-call check on this endpoint, with a review date of 11 October 2026. The memory-extraction client now retries once without `response_format` when it receives that specific 400, remembers the result for the lifetime of the process, parses the JSON itself, and fails closed if parsing fails; 4 new and 39 existing tests pass. The gateway's clients get a fast 400 in place of the risk of a hang. `tool_choice=required` is unavailable on this endpoint for as long as the flag stands.

The rollout itself caused a regression. From 10:36 to 11:25 UTC, the memory-extraction client's json_object calls received HTTP 400 and fell back silently to a hosted model until its client fix landed. We found this by checking callers after the rollout, not through an alert. For a fleet that runs its own inference, a silent fallback to a hosted model is a failure in its own right, even though the requests were answered.

We recorded three rollout lessons on the day, and they are listed here as recorded:

1. Scope post-restart checks to the new process invocation. A journal window that starts before the restart will read the old process's shutdown traceback. This cost us one false auto-revert and one false critical page.
2. After any server flag change, smoke-test a real completion, never the health endpoint alone, and watch the first minutes of logs. The crash-looping server died on its first decode.
3. Give every wait loop a deadline that reports failure. The silent until-loop turned a 4-minute failure into a 1 h 41 min unnoticed outage. The later rollout script had a 720 s deadline, a real completion as its pass condition, and automatic revert plus a page on failure.

Taken together, the three hangs cost about 11 to 12 minutes each, and the unverified flag change watched by a silent loop cost about 1 h 46 min. The remediation caused more outage than the bug did.

9. What These Numbers Will Not Carry

The counts are small. On the baseline server we have 2 of 2 harness hangs plus 1 production hang, one run of control A, and 2 runs of B that are known only from a summary, since their driver logs are not in the retained run directory. On the patched server we have 1 hang in 3 replicas, and two of those replicas lost concurrency partway through. For the fix we have one 10.4-minute S test, two short replicas, one 9-minute control A, and 22 minutes of monitored operation. Zero hangs over that window is strong support for a narrow claim: the incident request can no longer wedge the server, because by construction it never reaches decode. It is weak support for any general claim about the server's stability.

We tested one build on one topology. That is a development build of SGLang from a model-specific branch, TP4/EP4 across two nodes, DSpark block size 5, and xgrammar. We did not test a single-node deployment, other speculative methods, other grammar backends, or grammar with speculation off. We cannot say which of cross-node collectives, DSpark specifically, or xgrammar is necessary for the deadlock. The trigger we isolated ("a constrained request decoding in the same batch as another request") holds for this configuration only.

The root cause is not established. We know where the host waits in both scheduler modes, and we know the device spins. We do not know which collective mismatches or why. The accepted-length hypothesis is unverified. The claim that the deadlock is independent of scheduler mode rests on one patched hang, supported by the matching device signature.

The throughput numbers are thin. Each is a median of 3 at a single prompt shape, and the baseline figures come from a report rather than a retained raw file. A load test of the same shape on 26 September recorded 83 tok/s single-stream and 171 tok/s at concurrency 4 under different conditions. Numbers should be compared only within 27 September. The 6% and 4% cost of disabling overlap is an indication, not a characterisation.

The service history rests partly on summaries. The "12.5 h clean" and "hundreds of concurrent batches" figures come from summaries. The journal confirms about 12.3 h of the hung invocation without a watchdog fire.

We corrected several summary figures against the artefacts. Where the two disagreed, we followed the artefacts. The incident time is 05:41:46 UTC, not BST. There were 22 failed starts, not 21. Hang onset in C1 and C2 was 24 to 47 s after submission, not 1 to 3 min. The patched deadlock reproduced in 1 of 3 replicas, not 3/3.

The following claims survive:

- the combination of a grammar mask, a DSpark verify and concurrent batching deadlocks this deployment;
- utilisation alone does not reveal the failure, but power draw does;
- disabling overlap scheduling does not remove the deadlock;
- refusing constrained requests at admission removes the trigger without measurable throughput cost.

The following claims do not survive: any statement about other builds or topologies, any named root cause, and any claim that the fix makes the server generally stable beyond the 22 monitored minutes.

10. What We Changed, and What Comes Next

The endpoint now runs with `--grammar-backend none`, overlap scheduling and DSpark on, and constraint enforcement in the clients. Our rollout tooling now scopes checks to the new process invocation, passes only on a real completion, and gives every wait a deadline that reverts and pages.

We have drafted an upstream bug report covering the deadlock, the non-overlap TypeError, and the finding that the patched non-overlap path still deadlocks. It had not been filed at the time of writing. The draft suggests accepting `pp_proxy_tensors` in all v2 speculative workers. It also suggests that the overlap path either run the grammar barrier before the verify launch or fall back to non-overlap for batches containing a grammar, so that the host never waits on a stream holding a pending cross-rank collective. Our patched run suggests the fallback alone would not be enough.

If an upstream fix lands, we will remove the flag, restore the conformance probe's forced-tool check, and rerun the incident replica three times with the hang monitor armed before we trust the constrained path again.

PureTensor operates a sovereign AI infrastructure fleet (on-premises GPU compute, storage, and serving) run day-to-day by autonomous agents under human direction. This paper is PT-R-2026-029. The measurements were taken on 26 and 27 September 2026, with the incident, reproductions, both server-side attempts and the fix falling on 27 September between 05:41 and 11:25 UTC. The raw artefacts (service journals, reproduction driver logs, GPU captures and host stacks) are retained, apart from the driver logs for the constrained-alone runs noted above.