Resource Aware Attention:
A New Attention Mechanism Designed For CPUs

Adarsh M.S. Ditto P.S. Jithin V.G.
{adarsh.ms, dittops, jithinvg}@bud.studio

Version 1.0

April 2026

Introducing RAA, an attention mechanism designed against the resource envelope of the deployment target — and the first production results from a family of models built on it.

Abstract. A disproportionate share of every LLM deployment's latency and cost now lives in the small models on either side of the generative model — safety classifiers, routers, retrieval scorers, PII extractors. These models run on every request, almost always on CPU fleets, and they are almost always built from the same transformer template the generative model uses. That template was never designed for this role.

We introduce **Resource Aware Attention (RAA)**, a new attention mechanism designed from the ground up against the resource envelope of the deployment target — CPU cache hierarchy, precision tier, and latency SLO — with the envelope as a first-class input to the mechanism rather than a downstream compression concern. We also introduce the **RAA family**, the first line of models built under this design principle, and **BudSentinel**, the safety-guardrail product line built on the family's safety-specialized variant.

Across jailbreak and prompt-injection benchmarks, BudSentinel delivers **single-digit-millisecond classification on commodity CPUs** — faster than every transformer baseline we evaluated, including those same baselines running on an A100 GPU — while sustaining **over 4,400 requests per second on a single server-class Xeon node** at 512-token inputs, and **nearly 1,500 requests per second on a fanless laptop CPU**. It handles inputs as large as **65,536 tokens** on a 16-vCPU box — an input-length regime the transformer guard baselines, capped at 512 tokens, do not operate in at all. On accuracy, BudSentinel is the only model in our evaluation that lands both Attack Success Rate and False Refusal Rate under 20%, making it the only evaluated model that sits on a deployable operating point.

This paper describes the research direction behind RAA, its guiding design principles, the current family of models, headline benchmark results, honest trade-offs, applications well beyond guardrails, and what we plan to publish next.

---

1. Introduction

The modern LLM stack has quietly become a **pipeline of small models wrapped around a large one**. A single request to a production LLM is, in practice, a chain: a safety classifier decides whether the input is admissible; a router decides which generative model should see it; a retrieval scorer orders a candidate context set; a PII extractor redacts identifiers; a routing or policy classifier decides what to do with the output; another safety classifier checks the generation. The generative call is expensive, but most of what the user pays for — in wall-clock time and in cost per request — increasingly lives outside it.

These small models are almost all built from the same architectural template as the generative model: a transformer [1] trained on GPU and deployed by trying to compress it until it fits on a CPU. Existing safety guardrails [5, 8–12] all instantiate this template. The template works. It is also, we will argue, the wrong template for this role.

This paper introduces a different starting point. What if, instead of taking an architecture designed for a GPU and scaling it down, we designed an attention mechanism from the **resource envelope of the deployment target inward** — treating the cache hierarchy, the precision tier, and the latency SLO as inputs to the mechanism itself? We call the result **Resource Aware Attention (RAA)**, and we call the first line of models built on it the **RAA family**.

The headline result: BudSentinel, the first product from the RAA family, delivers jailbreak classification in **under 10 milliseconds on a laptop-class CPU** — faster than transformer competitors running on an A100 — with better deployable accuracy than any guard model we evaluated, and with a **native sequence-length range (up to 65,536 tokens) that the 512-token-capped transformer guard category does not reach at all.**

2. Why this is the right problem now

The small-model tax on the LLM stack has three compounding properties that make it urgent to solve at the mechanism level rather than the compression level.

The tax is per-request, not per-session. A generative call amortizes its cost across a long output. A safety classifier pays its cost once per input and again once per output. A retrieval scorer pays its cost once per candidate, many times per query. These workloads have no natural batching geometry the way an autoregressive decode does, and they cannot hide latency behind token streaming.

The deployment target is not a GPU. Operators do not put a dedicated GPU in front of every safety classifier; they cannot afford to, and the geometry of the call graph would not let them saturate it if they did. Safety, routing, and retrieval classifiers run on CPU fleets — often on the same nodes the application runs on, often on shared tenants, often with an L3 cache already under pressure from co-resident workloads. GPU attention's cost profile is simply the wrong profile for this deployment reality.

Compression has plateaued. The industry has leaned hard on the obvious tools: shrinking large transformers into small ones, quantizing activations and weights to int8, exporting to a faster runtime. These wins are real and well-established, but they have not moved the category out of the hundreds-of-milliseconds-per-request regime on CPU. Our evaluation shows the current generation of small transformer guard models landing between **334 ms and 3,854 ms per classification** on the CPUs operators actually run on. Another round of compression will not close that gap by two orders of magnitude.

And the input-length ceiling is a separate problem. Transformer guard models in this category almost universally cap input sequences at **512 tokens**, a direct consequence of the quadratic cost profile of standard self-attention [1] — beyond that length, the cost and memory become untenable on CPU. For workloads that need to score long transcripts, multi-turn conversation history, retrieved RAG context, uploaded documents,

or code files, the 512-token ceiling turns every long input into a separate engineering problem (chunking, voting, re-aggregation) that degrades both accuracy and latency further.

We believe the right move at this point is not to compress the existing mechanism further. It is to **design attention differently, with CPU inference as a first-class constraint of the mechanism itself**, and with long inputs as a native capability rather than a retrofit. That is the research direction RAA belongs to.

3. Resource Aware Attention

RAA is an attention mechanism whose design is governed by three principles.

The resource envelope is an input to the mechanism, not an output. The target cache hierarchy, the precision tier the runtime will execute in, and the latency SLO the application is committing to are all declared before training. The attention mechanism is defined against them. A model trained for one envelope is not retargeted to another through post-hoc compression; a new envelope is a new training run. This is the sense in which RAA is *resource aware*: the resources are not something the mechanism tolerates, they are something it is shaped by.

The attention step is bounded by the envelope, not by sequence length. RAA's attention does not suffer the quadratic growth of standard self-attention. A broader literature on efficient attention — Linformer [2], Performer [3], Longformer [4], and others — has shown that subquadratic attention is achievable, but those mechanisms were still designed first and resourced later. RAA inverts that order: long inputs cause the mechanism to do more of the same cheap work; they do not cause a phase change in cost. This is the structural reason RAA runs in single-digit milliseconds on CPU at 512-token inputs and in double-digit milliseconds at 16,384-token inputs, without the runtime ever being told it is "doing a long input."

Heads compose freely over a single attention pass. A single RAA attention pass supports many downstream task heads in parallel: a safety decision, a PII span extraction, an intent classification, a routing decision. The marginal cost of adding another head is the head's own work, not another attention pass. For systems that already chain five or six small classifiers per request, RAA turns that chain into one shared attention cost with many cheap decisions on top.

We want to be honest about the positioning RAA takes on the accuracy-versus-resource curve. RAA is not an attempt to match a frontier transformer, given unbounded compute, on every benchmark leaderboard. It is a deliberately different point on the curve — and the curve looks fundamentally different once the resource envelope is built into the mechanism rather than bolted on afterward. The rest of this paper makes that case with numbers.

4. The RAA family of models

RAA is a mechanism. The **RAA family** is the first line of models built on it. We deliberately organize the family by the *task class* and *resource envelope* each variant is designed for, rather than by parameter count.

- **RAA-Base.** The general-purpose backbone of the family. Targets a standard server-CPU envelope and supports classification, scoring, span extraction, and retrieval-style tasks with equal first-class standing. The foundation most of the family's task-specific variants are built from.
- **RAA-Safety.** The safety-specialized variant. Trained against the jailbreak, prompt-injection, and content-moderation surfaces that sit on the input and output edges of an LLM. Powers the **BudSentinel** product line.
- **RAA-Span (roadmap).** A token-level and span-level variant tuned for PII extraction, compliance tagging,

code detection, and secret scanning — workloads that need attribution, not just a score.

- **RAA-Retrieve (roadmap).** A retrieval-oriented variant for re-ranking and scoring at candidate-set scale, designed for the resource envelope of a retrieval pipeline rather than a classifier.
- **RAA-Route (roadmap).** A routing-oriented variant for request classification at the edge of a model gateway, targeting a tighter envelope than RAA-Base and a sub-millisecond SLO.

The family is explicitly designed to be **run together**. Because every RAA variant shares the same mechanism, multiple variants can be co-hosted in a single process and share a single attention pass per request. A gateway that today runs a separate safety transformer, router transformer, and PII transformer — each at 100–900 ms per request on CPU — can in principle replace all three with a single-process RAA deployment whose per-request cost is dominated by one attention pass, regardless of how many heads it carries.

5. BudSentinel: the first product built on RAA

BudSentinel is the safety-guardrail product line built on RAA-Safety. Its scope covers jailbreak detection, prompt-injection detection, and content moderation — toxicity, hate, harassment, self-harm, violence, illegal content, and regulated advice categories.

BudSentinel is designed to be deployed *between* a user and a model, on the same fleet the application already runs on, at the latency budget the application already has. It is not designed to live in a separate GPU service with its own scaling story. A safety classifier that adds 300 ms to every turn changes the shape of an application; a safety classifier that adds 8 ms does not. The rest of this paper is the case for why that difference matters, and how far it generalizes beyond safety.

6. Benchmark results

All numbers in this section are measured against public benchmark sets and public competitor models. Methodology and full tables are in the appendix.

6.1 Accuracy on jailbreak and prompt-injection

We evaluate on an aggregated benchmark drawn from four widely used jailbreak / prompt-injection suites: JailBreakBench [6], PIGuard [8], WildJailbreak [7], and the Qualifire Prompt Injection Benchmark [13]. We report **Attack Success Rate (ASR)** — the probability a harmful prompt gets through — and **False Refusal Rate (FRR)** — the probability a benign prompt is incorrectly blocked.

Table 1: Overall aggregate accuracy across all four benchmarks.

Model	ASR	FRR
BudSentinel-Jailbreak	**15.97%**	**14.92%**
Prompt-Guard-2-86M	34.68%	15.30%
Prompt-Guard-86M	5.83%	89.35%
ArchGuard	5.40%	81.65%
PIGuard	25.01%	25.86%
ProtectAI PI V2	36.35%	24.39%

BudSentinel-Jailbreak is **the only model in the evaluation with both ASR and FRR below 20%**. The two models that show lower ASR numbers (Prompt-Guard-86M, ArchGuard) achieve them by refusing almost everything — 89% and 82% false-refusal rates, respectively. No product team can ship those operating points.

We think the metric that matters is the **deployable operating point** — an (ASR, FRR) pair a product can actually run in production. On the deployable frontier, BudSentinel-Jailbreak is not merely the leader; in our evaluation, it is the only occupant.

On the harder individual slices, the same pattern holds, with the strongest single result on **WildJailbreak**: **ASR 6.5%, FRR 11.42%** — simultaneously lower false-negative and lower false-positive rates than every other evaluated model on that benchmark. The PIGuard slice is the hardest for every model we evaluated, including ours; the ordering on the deployable frontier is preserved but absolute ASR rises across the board. We discuss this in §7.

6.2 Per-request latency on CPU

All models evaluated at 512-token sequence length — which is the *maximum* sequence length most of the transformer baselines support. Competitor baselines measured via their published runtime pipelines on the bare machine; BudSentinel measured end-to-end over its serving interface, so the measurement includes the network and RPC path that a real client request would traverse.

Table 2: Per-request latency at 512-token sequence length across CPU and GPU targets.

Model	i7-11370H (laptop)	A100-80GB (GPU)	EPYC 7V13	Xeon 8272CL
BudSentinel-Jailbreak	**8.39 ms**	n/a	**5.67 ms**	**5.99 ms**
Prompt-Guard-2-86M	803.85 ms	18.52 ms	3854.91 ms	334.15 ms
Prompt-Guard-86M	883.99 ms	18.92 ms	3871.68 ms	401.97 ms
ArchGuard	850.43 ms	19.07 ms	3851.31 ms	379.89 ms
PIGuard	844.18 ms	19.00 ms	3777.85 ms	349.33 ms
ProtectAI PI V2	841.37 ms	19.13 ms	3842.30 ms	345.19 ms

Two observations worth pulling out:

1. **BudSentinel on a laptop CPU is faster than every transformer peer running on an A100.** Not comparable — *faster*. The A100 numbers are 18–19 ms; BudSentinel on an i7-11370H is 8.39 ms end-to-end including the RPC path.
2. **On CPU, the gap is one-to-three orders of magnitude.** The transformer baselines on EPYC 7V13 sit between 3.7 and 3.9 seconds per classification. BudSentinel on the same CPU class lands at 5.67 ms.

6.3 Production throughput on CPU

Per-request latency is only part of the story. The deployment question operators ask is: what does a single node actually sustain under load, across the envelopes teams actually deploy into? We report three: a modern server-class Xeon, a commodity server-class Xeon for long-sequence workloads, and a laptop-class fanless CPU to characterize the on-device end of the envelope.

Server class — Intel Xeon 6972P:

Input	Concurrency	p50 (ms)	p95 (ms)	p99 (ms)	Throughput (req/s)
128 tokens	1	2.57	3.04	3.11	341
128 tokens	10	3.99	5.03	5.25	1,937
128 tokens	50	7.00	10.45	11.35	**4,690**
512 tokens	1	3.05	3.25	3.28	330
512 tokens	10	4.57	5.30	5.59	1,809
512 tokens	50	7.96	10.78	11.16	**4,445**

At 50-way concurrency, a single Xeon 6972P sustains **over 4,400 requests per second at 512-token inputs** with a **p99 below 12 ms** — numbers that translate directly into "you can put a safety classifier on every request in a high-traffic LLM gateway and it will never be your bottleneck."

Long-sequence class — Intel Xeon Platinum 8272CL:

Seq Length	Concurrency	p10 (ms)	p50 (ms)	p90 (ms)	p99 (ms)	Throughput (req/s)
512	100	9.29	25.17	34.93	44.16	2,704
8,192	100	67.05	151.07	214.06	262.75	437
16,384	100	97.97	254.73	404.86	571.22	235
65,536	100	197.00	559.98	933.79	1214.18	100

Two things stand out. First, on a commodity box BudSentinel sustains **over 2,700 req/s at 512-token inputs** with a 25 ms p50. Second, and more importantly: **the same model classifies 65,536-token inputs at 560 ms p50 on that same box.** The transformer guard baselines in §6.2 do not operate at this input length at all — their architecture caps them at 512 tokens, and production deployments that need to score longer inputs must resort to chunking, voting, and re-aggregation pipelines that cost latency, cost accuracy, and add engineering surface. BudSentinel classifies the entire input in a single call.

Laptop / on-device class — Intel Core Ultra 7 268V (Lunar Lake):

Input	Concurrency	p50 (ms)	Throughput (req/s)
128 tokens	1	1.18	746
128 tokens	50	22.99	**1,493**
512 tokens	1	1.80	498
512 tokens	50	26.32	881

On a fanless ultraportable laptop CPU, BudSentinel sustains **nearly 1,500 requests per second** at 128 tokens and over 880 at 512 tokens. This is the on-device story: a safety classifier that can plausibly run inside a desktop application, a browser extension, an IDE, or an embedded agent, without calling out to a server at all.

A small methodology note: BudSentinel's latency figures throughout this section include the full serving path to the model (network, RPC, scheduling), while the transformer baselines in §6.2 were measured locally with no such overhead. The comparison is, if anything, conservative in BudSentinel's favor.

7. Accuracy and safety trade-offs

We want to be explicit about what RAA is not.

RAA is not generative. It is a mechanism for classification, scoring, and span extraction. It is not a replacement for a frontier LLM on open-ended reasoning, and it is not trying to be.

RAA trades expressive ceiling for resource discipline. On tasks where a top-tier transformer, given unbounded compute, would eke out a small absolute-accuracy edge over RAA, RAA will not match it. That is a conscious design choice: the mechanism exists to deliver the most accurate *deployable* model at the target resource envelope, not the most accurate model at any compute cost. On every benchmark in §6.1, the models that outperform BudSentinel on raw ASR do so only at a false-refusal rate no product can live with — which is exactly the asymmetry the resource envelope framing is meant to expose.

Harder slices are harder for everyone, including us. On the PIGuard benchmark slice, absolute ASR rises across every evaluated model, ours included. The deployable ordering holds, but we would rather say so directly than paper over it. Security classification is an adversarial field, and we expect the benchmark floor to keep moving.

The RAA layer is one of several in BudSentinel. A fast safety classifier is an excellent *first* line of defense — cheap enough to run on every request, accurate enough to catch the bulk of what matters — but we do not ask it to be the whole defense. BudSentinel ships as a layered product. The RAA-powered fast layer handles the firehose at single-digit-millisecond latency; behind it sit deeper scans — higher-capacity review for tail risk, specialized detectors for adversarial and long-horizon patterns, and escalation paths for ambiguous cases. Customers integrate the full stack, not just the fast classifier. The speed of the fast layer is what makes the deeper layers affordable to run at production volume.

8. Applications beyond guardrails

RAA is an attention mechanism, not a guardrail trick. Any CPU-deployed small-model workload in the modern LLM stack is a candidate. We expect the RAA family to see use across the categories below.

Safety and moderation. The flagship application, and the shape of the story BudSentinel tells: put a classifier on every input and output of every LLM call without changing the application's latency or cost profile.

Retrieval and re-ranking. Retrieval pipelines score thousands of candidate passages per query. Doing that on CPU within the latency budget of a gateway is a textbook RAA workload — linear scaling in input length, multi-head composition over a single pass, no dependence on a dedicated GPU.

Routing and model selection. At the edge of a model gateway, "which generative model should handle this request?" is a small-model decision that every request pays for. A millisecond-class RAA router is strictly dominant over a hundred-millisecond transformer router; most gateways simply cannot afford the latter.

Entity, PII, and compliance extraction. Span-level tagging is required on virtually every request in regulated deployments. An RAA span variant will let this happen on the same node as the application, without a per-request GPU hop, and alongside the safety pass that was already going to run.

Query understanding. Search and agent stacks classify, rewrite, and route queries under tight budgets. These stacks were built around existing CPU models; RAA variants fit the envelope already paid for and leave room for more decisions per query.

Long-context classification, scoring, and extraction. Uploaded documents, multi-turn conversation histories, retrieved RAG contexts, full code files, meeting transcripts — every one of these is a long-input workload that the 512-token-capped transformer classifier category handles only via chunking, voting, and re-aggregation pipelines, losing both accuracy and latency in the process. RAA handles inputs up to 65,536 tokens natively, so long-context safety, PII, compliance, and scoring decisions are a single classifier call instead of an engineering project.

Edge and on-device. The resource envelope can be dialed well below a server CPU. On a fanless Lunar Lake laptop CPU, BudSentinel already sustains nearly 1,500 req/s at 128 tokens. That unlocks guardrails inside a desktop application, a browser extension, an IDE, an offline assistant, or an embedded industrial

gateway — targets where a transformer in the classic sense does not fit at all. RAA's design starts from an envelope declaration, so tighter targets are a natural part of the family, not a compression exercise.

The through-line across these categories is the same: when the attention mechanism is designed against the envelope the application actually deploys into, the set of decisions the application can afford to make per request goes up. RAA is a bet that this change — more decisions, faster, at the same budget — is architecturally valuable, not incidentally so.

9. Research directions

We see three active directions for the RAA family over the next year.

More variants, more envelopes. We will publish RAA variants targeted at progressively tighter envelopes (single-core, mobile, embedded) and progressively richer tasks (retrieval, multi-hop classification, span linking). The goal is to make "what envelope are you deploying into?" a question a team can answer by picking a family member, not by running a compression project.

Multi-variant co-hosting. Because every RAA variant shares the same mechanism, we are actively working on running several variants together in a single process over a shared attention pass. The end state is a single-process deployment that carries a full guardrail, routing, and span-extraction stack for the same per-request cost as any one of them today.

Evaluation infrastructure. Jailbreak and prompt-injection benchmarks age fast, and the deployable-operating-point framing we use in §6.1 is underrepresented in the literature. We are committed to publishing harder, less-gameable versions of these benchmarks, updating these numbers as the field moves, and being direct about regressions when they happen.

10. Closing

The resource envelope has been an afterthought in attention design for too long. Model architectures are chosen on GPUs and taped together with compression tricks on their way to CPU. In most of the modern LLM stack — in every small model on every request — that is the wrong order of operations.

RAA is a bet on a different order. Declare the envelope first. Design the attention mechanism against it. Build a family of models that share the mechanism, not a family of compressed transformers that share a template. The numbers in this paper are the first evidence that the bet is paying off; the roadmap in §9 is what we plan to do next.

We would like to talk to teams operating at the edges where this matters — safety pipelines, retrieval stacks, gateways, on-device deployments. If that is you, please reach out.

Appendix: Full benchmark tables

Benchmark methodology

All competitor models were evaluated through their published inference pipelines running locally on the measurement host. BudSentinel was evaluated end-to-end over its serving interface, so its latency numbers include the full network and RPC path a real client request would traverse. All models were evaluated at 512-token sequence length unless otherwise noted. Accuracy metrics follow the standard definitions: ASR is the false-negative rate on the harmful class; FRR is the false-positive rate on the benign class.

A.1 Jailbreak and prompt-injection accuracy — full tables

Overall aggregate

Model	ASR	FRR
BudSentinel-Jailbreak	15.97%	14.92%
Prompt-Guard-2-86M	34.68%	15.30%
Prompt-Guard-86M	5.83%	89.35%
ArchGuard	5.40%	81.65%
PIGuard	25.01%	25.86%
ProtectAI PI V2	36.35%	24.39%

JailBreakBench (Benign: 100, Non-Benign: 1,900)

Model	ASR	FRR
BudSentinel-Jailbreak	16.20%	47.00%
Prompt-Guard-2-86M	27.94%	17.00%
Prompt-Guard-86M	2.63%	98.00%
ArchGuard	1.00%	100.00%
PIGuard	18.36%	26.00%
ProtectAI PI V2	34.10%	1.00%

PIGuard (Benign: 1,310, Non-Benign: 125)

Model	ASR	FRR
BudSentinel-Jailbreak	50.40%	14.96%
Prompt-Guard-2-86M	99.20%	12.29%
Prompt-Guard-86M	36.00%	73.20%
ArchGuard	13.60%	87.40%
PIGuard	37.60%	20.68%
ProtectAI PI V2	89.60%	29.46%

WildJailbreak (Benign: 210, Non-Benign: 2,000)

Model	ASR	FRR
BudSentinel-Jailbreak	6.50%	11.42%
Prompt-Guard-2-86M	56.30%	28.50%
Prompt-Guard-86M	8.50%	91.40%
ArchGuard	14.20%	72.38%
PIGuard	39.25%	36.19%
ProtectAI PI V2	40.20%	32.85%

Qualifire Prompt Injection Bench (Benign: 3,001, Non-Benign: 1,999)

Model	ASR	FRR
BudSentinel-Jailbreak	18.20%	5.56%
Prompt-Guard-2-86M	36.90%	14.36%
Prompt-Guard-86M	7.90%	93.90%
ArchGuard	9.65%	75.14%
PIGuard	32.60%	22.89%
ProtectAI PI V2	34.40%	23.40%

A.2 Per-request latency

Measured at 512-token sequence length. BudSentinel measured end-to-end over its serving interface; competitors measured via their published inference pipelines locally.

Model	i7-11370H @ 3.30 GHz	A100-SXM4 -80GB	EPYC 7V13 64-Core	Xeon 8272CL @ 2.60 GHz
BudSentinel-Jailbreak	8.39 ms	n/a	5.67 ms	5.99 ms
Prompt-Guard-2-86M	803.85 ms	18.52 ms	3854.91 ms	334.15 ms
Prompt-Guard-86M	883.99 ms	18.92 ms	3871.68 ms	401.97 ms
ArchGuard	850.43 ms	19.07 ms	3851.31 ms	379.89 ms
PIGuard	844.18 ms	19.00 ms	3777.85 ms	349.33 ms
ProtectAI PI V2	841.37 ms	19.13 ms	3842.30 ms	345.19 ms

A.3 Production throughput — Intel Xeon 6972P

Input	Concurrency	p50 (ms)	p95 (ms)	p99 (ms)	Throughput (req/s)	Error Rate
128 tokens	1	2.57	3.04	3.11	341	0.00%
128 tokens	10	3.99	5.03	5.25	1,937	0.00%
128 tokens	50	7.00	10.45	11.35	4,690	0.00%
512 tokens	1	3.05	3.25	3.28	330	0.00%
512 tokens	10	4.57	5.30	5.59	1,809	0.00%
512 tokens	50	7.96	10.78	11.16	4,445	0.00%

A.4 Production throughput — Intel Xeon Platinum 8272CL (long-sequence characterization)

Seq Len	Concurrency	p10 (ms)	p50 (ms)	p90 (ms)	p99 (ms)	Throughput (req/s)
512	100	9.29	25.17	34.93	44.16	2,704
512	5,000	267.17	1,027.03	1,732.37	1,867.63	2,674
512	10,000	542.36	1,910.90	3,340.36	3,664.11	2,749
8,192	100	67.05	151.07	214.06	262.75	437
8,192	5,000	1,093.94	4,968.95	8,809.76	9,737.09	509
8,192	10,000	2,146.64	9,935.21	17,664.37	19,434.16	508
16,384	100	97.97	254.73	404.86	571.22	235
16,384	10,000	3,697.23	17,989.47	31,802.07	34,860.60	283
65,536	100	197.00	559.98	933.79	1,214.18	100
65,536	10,000	10,479.62	50,874.37	88,964.59	97,428.30	101

A.5 Production throughput — Intel Core Ultra 7 268V (Lunar Lake)

Input	Concurrency	p50 (ms)	Throughput (req/s)
128 tokens	1 (low load)	1.18	746
128 tokens	50 (high load)	22.99	1,493
512 tokens	1 (low load)	1.80	498
512 tokens	50 (high load)	26.32	881

References

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017. arXiv:1706.03762.

[2] S. Wang, B. Z. Li, M. Khabsa, H. Fang, and H. Ma. Linformer: Self-attention with linear complexity. *arXiv preprint* arXiv:2006.04768, 2020.

[3] K. Choromanski, V. Likhosherstov, D. Dohan, X. Song, A. Gane, T. Sarlos, P. Hawkins, J. Davis, A. Mohiuddin, L. Kaiser, D. Belanger, L. Colwell, and A. Weller. Rethinking attention with Performers. In *International Conference on Learning Representations (ICLR)*, 2021. arXiv:2009.14794.

[4] I. Beltagy, M. E. Peters, and A. Cohan. Longformer: The long-document transformer. *arXiv preprint* arXiv:2004.05150, 2020.

[5] H. Inan, K. Upasani, J. Chi, R. Rungta, K. Iyer, Y. Mao, M. Tontchev, Q. Hu, B. Fuller, D. Testuggine, and M. Khabsa. Llama Guard: LLM-based input-output safeguard for human-AI conversations. *arXiv preprint* arXiv:2312.06674, 2023.

[6] P. Chao, E. Debenedetti, A. Robey, M. Andriushchenko, F. Croce, V. Sehwag, E. Dobriban, N. Flammarion, G. J. Pappas, F. Tramèr, H. Hassani, and E. Wong. JailbreakBench: An open robustness benchmark for jailbreaking large language models. In *NeurIPS Datasets and Benchmarks Track*, 2024. arXiv:2404.01318.

[7] L. Jiang, K. Rao, S. Han, A. Ettinger, F. Brahman, S. Kumar, N. Mireshghallah, X. Lu, M. Sap, Y. Choi, and N. Dziri. WildTeaming at scale: From in-the-wild jailbreaks to (adversarially) safer language models. *arXiv preprint* arXiv:2406.18510, 2024.

[8] H. Li, X. Liu, N. Zhang, and C. Xiao. PIGuard (InjecGuard): Prompt injection guardrail via mitigating overdefense for free. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (ACL)*, 2025. arXiv:2410.22770.

[9] Meta AI. Prompt-Guard-86M. Model card, 2024. https://huggingface.co/meta-llama/Prompt-Guard-86M.

[10] Meta AI. Llama Prompt Guard 2 (86M and 22M). Model card, 2025. https://huggingface.co/meta-llama/Llama-Prompt-Guard-2-86M.

[11] Katanemo Labs. Arch-Guard: Jailbreak detection model. Model card, 2024. https://huggingface.co/katanemo/Arch-Guard.

[12] ProtectAI. DeBERTa-v3-base prompt injection v2. Model card, 2024. https://huggingface.co/protectai/deberta-v3-base-prompt-injection-v2.

[13] Qualifire. Prompt Injection Benchmark. Dataset, 2024. https://huggingface.co/datasets/qualifire/prompt-injections-benchmark.