Benchmarking Single-Card LLM Inference on Intel Gaudi 3 for Enterprise Workloads

1. Summary

As generative AI systems move from experimentation into production, inference becomes the dominant workload. For most organizations, the central challenge is no longer model accuracy alone, but how to deliver large language model responses at scale while maintaining predictable latency, sustainable infrastructure costs, and operational stability.

This whitepaper is intended for enterprise architects, platform engineers, and technical decision-makers evaluating inference infrastructure options. It addresses a practical question: how does Intel Gaudi 3 perform for real-world inference workloads when deployed on Red Hat OpenShift AI, and what does that performance mean for cost, capacity planning, and production readiness?

To answer this, AHEAD partnered with Intel to conduct structured, repeatable benchmarking of large language model inference on Gaudi 3 accelerators within an OpenShift AI environment. The goal was not to produce a single peak performance number, but to evaluate how throughput, latency, and concurrency behave under realistic conditions, including varying prompt sizes and increasing concurrent request volumes.

Across small to mid-sized open-weight models, Gaudi 3 demonstrated strong single-card throughput and efficient concurrency scaling within practical operating ranges. For many latency-sensitive inference workloads, the platform sustained predictable performance up to defined concurrency inflection points, after which saturation behavior was clearly observable. These characteristics are directly relevant to cost modeling and capacity planning, as they help teams determine how many accelerators are required to meet sustained traffic demands without overprovisioning.

While this paper does not claim universal performance leadership across all models and workloads, it provides transparent, reproducible data that can inform price/performance evaluation and architectural decision making. By grounding the analysis in operational metrics rather than synthetic peaks, the results are intended to help organizations assess where Gaudi 3 aligns with their inference profile and business objectives.

Readers seeking a concise interpretation can focus on the summary tables and benchmark analysis sections. Those interested in full technical transparency, including prompt definitions, token accounting, runtime configuration, and metric collection methodology, can reference the appendix for complete reproducibility details.

The results presented in this paper are intended to characterize the behavior of a single Intel Gaudi 3 accelerator under controlled inference workloads. They should be interpreted within the following scope:

- **Single-accelerator evaluation:** All results reflect performance on a single Gaudi 3 device. Multi-accelerator scaling, distributed inference, and interconnect behavior are not evaluated.
- **Inference-focused scope:** This paper evaluates inference serving performance only. Model training, fine-tuning, and large-scale distributed training scenarios are out of scope.
- **Model class and size range:** The evaluation focuses on small to mid-sized open-weight models commonly used in enterprise serving environments. Results may not generalize to frontier-scale models.
- **Workload and runtime specificity:** Observations are based on the specific combination of vLLM, Red Hat OpenShift AI, and the configurations described in this paper. Performance characteristics may differ under alternate runtimes, frameworks, or system configurations.
- **Non-comparative positioning:** This paper does not present cross-vendor benchmarking or claims of universal performance leadership. Instead, it focuses on understanding operating characteristics within a defined environment.

2. Benchmark Overview

2.1 The Datasets We Used

The benchmarks were designed to reflect realistic inference serving conditions rather than synthetic micro-tests. Instead of relying on a single fixed prompt, we evaluated performance across a structured set of prompts that varied in input token length while maintaining consistent task structure.

This approach allowed us to observe how throughput and latency change as prompt size increases and as concurrency scales, which more closely reflects production behavior where request sizes are not uniform.

The full prompt set, token counts, and configuration details are documented in the appendix for transparency and reproducibility.

2.2 The Models We Tested

The benchmark was designed to reflect realistic inference in serving conditions rather than synthetic peak tests. Performance was evaluated across a structured prompt set with progressively increasing input lengths and controlled output sizes to simulate common production patterns such as chat, summarization, and reasoning workloads. Full prompt text and token accounting are provided in the appendix.

Inference was tested using the following widely adopted open-weight models:

- Mistral 7B
- Qwen 2.5 7B
- Qwen 2.5 14B

These models were selected to represent commonly deployed small and mid-sized language models used in latency-sensitive inference scenarios. They vary in parameter count and computational characteristics while remaining practical for real-world enterprise deployment.

All benchmarks were executed using a consistent serving configuration to ensure comparability across models.

All results presented in this paper reflect single-card inference performance on Intel Gaudi 3. The analysis is intentionally scoped to single-accelerator behavior to isolate throughput, latency, and concurrency characteristics without multi-card scaling effects.

The purpose of testing multiple models within the same framework is to illustrate how inference performance scales with model size and to help identify practical operating ranges for production deployments.

2.3 Metrics and Business Impact

To evaluate inference behavior in a way that supports real architectural decisions, the benchmark focused on three core metrics: throughput, latency, and concurrency. Each metric was selected not only for technical relevance, but for its direct connection to business outcomes.

- **Throughput:** Throughput measures how many tokens are generated per second by a single accelerator. In this document, throughput refers specifically to completion token throughput (tokens generated by the

model), not total token throughput (prompt + completion), unless otherwise stated. This distinction ensures consistent comparisons across workloads with varying prompt lengths. From a business perspective, throughput directly impacts infrastructure efficiency and cost. Higher sustained throughput reduces the number of accelerators required to meet steady-state demand, influencing total cost of ownership and long-term scalability planning.

- **Latency:** Latency measures how quickly responses are returned to users. Both median and tail latency matter. Median latency shapes the typical user experience, while tail latency determines SLA reliability and worst-case behavior under load. Stable latency at higher concurrency levels supports customer-facing AI applications where responsiveness and predictability are critical.
- **Concurrency:** Concurrency reflects how many simultaneous inference requests are actively being processed by the serving system at a given time. In this benchmark, concurrency was controlled by issuing parallel requests to the vLLM endpoint using an OpenAI-compatible API, with each request representing an independent completion task. It is important to distinguish concurrency from batch size. Concurrency refers to the number of in-flight requests at the system level, while batching is an internal optimization handled by the runtime. In this setup, vLLM dynamically schedules and batches incoming requests to maximize accelerator utilization. As concurrency increases, the system can process more requests in parallel, improving overall throughput up to a saturation point. Beyond that point, additional concurrency leads to queueing, resource contention, and increased latency without corresponding throughput gains. From a business perspective, concurrency defines the practical capacity of a single accelerator. It informs how many user requests can be served simultaneously while maintaining acceptable latency and predictable performance.

Taken together, these metrics enable informed trade-offs between cost, performance, and reliability. Rather than optimizing for a single peak number, the analysis identifies practical operating ranges that balance efficiency with predictable behavior. This approach aligns with AHEAD's AI platform principles: build solutions that are measurable, operationally realistic, and economically sustainable for long-term production deployment.

2.4 Translating Performance to Capacity and Cost

While throughput, latency, and concurrency describe how an inference system behaves, enterprise teams ultimately need to translate these metrics into capacity requirements and cost expectations. This requires connecting observed performance to a simple, repeatable unit economics model.

At a high level, inference capacity planning can be expressed in terms of tokens processed per second and the number of accelerators required to sustain a target workload.

To estimate capacity requirements, teams should define the following workload characteristics:

- **Queries per second (QPS):** average number of incoming requests
- **Average output tokens per request:** typical response length
- **Target utilization:** desired operating range for each accelerator (commonly 70–80% to preserve headroom and latency stability)
- **Sustained tokens per second per accelerator**: measured performance within a stable concurrency band, not peak throughput

Using these inputs, capacity can be estimated using the following relationships:

- Required tokens per second = QPS × average output tokens
- Effective tokens per second per accelerator = sustained tokens per second × utilization target
- Accelerators required = required tokens per second / effective tokens per second per accelerator

For example, consider a workload with the following characteristics:

- 20 QPS
- 300 average output tokens per request
- Target utilization of 75%
- Measured sustained throughput of 4,000 tokens per second per accelerator within a stable operating range

The required throughput is:

- 20 × 300 = 6,000 tokens per second

Effective throughput per accelerator at target utilization:

- 4,000 × 0.75 = 3,000 tokens per second

Estimated accelerators required:

- 6,000 / 3,000 = 2 accelerators

In practice, teams would add additional headroom to account for burst traffic, variability in prompt length, and tail latency requirements.

This model provides a direct bridge between benchmark results and real deployment decisions. Rather than relying on peak throughput numbers, teams can use sustained performance within known operating ranges to estimate infrastructure requirements, compare deployment options, and plan for predictable scaling.

3. Developer Experience

3.1 The Benefits of Using OpenShift AI with Gaudi

From a developer perspective, working with Intel Gaudi 3 accelerators on Red Hat OpenShift AI was notably straightforward once the environment was up and running. Compared to other accelerator-based lab environments, the overall experience was more consistent and required less manual intervention.

A key reason for this is the amount of foundational integration work that has already occurred between Intel, Red Hat, and the open-source ecosystem. Gaudi support is not layered on as an afterthought. It is backed by a mature software stack, including Intel's Gaudi software suite, a Gaudi-optimized fork of vLLM, and Red Hat–maintained model and serving integrations that are designed to work cleanly within OpenShift AI.

The Gaudi-optimized vLLM fork provides native support for Gaudi hardware, exposing performance-critical features such as HPU Graphs, optimized KV cache handling, tensor parallelism, and FP8 inference through familiar configuration patterns. This allows developers to run inference workloads on Gaudi using the same serving abstractions and workflows they would expect when working with vLLM in other environments, without requiring application-level changes or custom kernels.

On the OpenShift side, Gaudi accelerators integrate naturally into the platform's containerized and Kubernetes-native model. Serving runtimes, resource scheduling, and workload management behave as expected, and Gaudi-specific considerations are handled largely at the runtime and container level rather than pushed onto the application developer. This separation of concerns makes it easier to iterate quickly once the environment is operational.

The initial setup did involve a small number of expected hurdles, such as initial resource permissions and role bindings, primarily related to environment readiness and configuration alignment. These issues were minor in scope and typical of working in shared or enterprise lab environments. Importantly, they were not specific to Gaudi itself and did not require deep hardware-level debugging or custom workarounds. Once resolved, the environment behaved predictably and remained stable throughout the benchmarking process.

After the initial setup phase, the workflow felt seamless. Model serving, configuration changes, and iterative benchmarking runs could be performed without repeatedly revisiting low-level system concerns. This made it possible to focus on benchmarking behavior, tuning parameters, and analyzing results rather than managing infrastructure friction.

This experience stands in contrast to other accelerator environments where developers often spend a disproportionate amount of time addressing driver mismatches, dependency conflicts, or opaque performance issues before meaningful experimentation can begin. In this case, the combination of Gaudi's software maturity, the vLLM integration, and OpenShift AI's serving abstractions resulted in a fast path from setup to productive iteration.

From an enterprise perspective, this matters. A smoother developer experience reduces time to value, lowers the operational burden on engineering teams, and makes it easier to reproduce results across environments. For teams evaluating inference platforms, the ability to move quickly from environment setup to reliable, repeatable benchmarking and deployment is just as important as raw performance metrics.

3.2 How to Replicate This Benchmark

The benchmarking workflow used in this analysis was designed to be repeatable and representative of how teams would deploy and evaluate inference workloads in an enterprise OpenShift AI environment. Rather than relying on ad hoc scripts or one-off configurations, the process followed a structured approach that maps closely to standard OpenShift AI model serving practices.

At a high level, the workflow consists of preparing a Gaudi-enabled OpenShift AI environment, deploying a Gaudi-optimized vLLM serving runtime, and executing controlled inference benchmarks across a defined set of models, prompt sizes, and concurrency levels. The goal was to ensure that results could be reproduced without requiring deep hardware-specific customization or bespoke tooling.

The benchmark flow can be summarized as follows:

- Provision an OpenShift AI environment with Gaudi 3 accelerators available to worker nodes.
- Configure a serving runtime based on the Gaudi-optimized vLLM fork, using container images aligned with the installed Gaudi software stack.
- Deploy models using OpenShift AI model serving constructs, ensuring consistent runtime and resource configuration across runs.
- Execute inference benchmarks by varying prompt size, output length, and concurrency parameters in a controlled and repeatable manner.
- Collect throughput and latency metrics for analysis across different operating points.

From a developer standpoint, most of the complexity is handled at the runtime and platform layer rather than in application code. Benchmark parameters such as prompt length, concurrency, and batching behavior are controlled through configuration rather than code changes, which makes it easy to iterate and explore different performance profiles.

While this paper focuses on the results and high-level workflow, detailed configuration parameters, prompt definitions, and token counts are documented in the appendix.

In practice, once the environment is prepared, the process of running additional benchmarks or testing new models is straightforward. This makes it feasible to adapt the same workflow to customer-specific workloads, different traffic patterns, or alternative operating assumptions without redesigning the benchmarking framework.

Refer to Jupyter Notebooks for ready-to-use code.

3.3 System Requirements

The benchmarks in this analysis were executed in a controlled Red Hat OpenShift AI environment configured specifically for single-card inference workloads on Intel Gaudi 3 accelerators. The goal was to maintain a consistent serving configuration across models so that observed differences could be attributed to model size, prompt characteristics, and concurrency behavior rather than environmental drift.

Hardware and Infrastructure Requirements

- Intel Gaudi 3 accelerators available on OpenShift worker nodes
- Single-accelerator configuration per benchmark run

- Sufficient system memory and storage to accommodate model weights, KV cache allocation, and benchmark artifacts
- Persistent storage for both notebook workspace state and model / results data

Two persistent volume claims were used during testing:

- Workbench PVC mounted at /opt/app-root/src for notebooks and workspace state
- Model and Results PVC mounted at /mnt/models for model weights, logs, and benchmark outputs

All results presented in this analysis are based on single-card inference performance. This was an intentional choice to isolate accelerator behavior and simplify interpretation of throughput, latency, and concurrency trade-offs.

Software and Platform Requirements

- Red Hat OpenShift with OpenShift AI installed and configured for model serving
- KServe serverless enabled
- Service Mesh enabled and configured
- Gaudi software stack aligned with the deployed accelerator firmware and drivers
- Gaudi-optimized vLLM serving runtime built from the maintained Gaudi vLLM fork
- Container images compatible with the installed Gaudi software version
- Prometheus monitoring stack with Thanos querier available for utilization collection
- Access to model artifacts via supported registries or object storage

All models were served using a consistent deployment pattern:

- Gaudi-enabled vLLM ServingRuntime
- Single-model-per-pod configuration
- PVC-based model loading
- OpenAI-compatible vLLM API endpoints

Within the serving runtime, Gaudi devices were exposed through habana.ai/gaudi, vLLM was configured for HPU execution, HPU Graph warmup was enabled, and resource configuration was kept consistent across models. Warmup behavior was expected and observed, particularly forlarger models.

Execution Workflow Assumptions

Benchmark execution followed a repeatable notebook-driven process. The environment was prepared once per project, models were deployed once per model, and benchmark runs were then executed by varying prompt size, output length, and concurrency according to a predefined checklist. Throughput and latency were measured at the vLLM API layer, while Gaudi utilization was collected from Prometheus through the in-cluster Thanos querier.

These requirements are representative of an enterprise OpenShift AI inference environment and are sufficient to reproduce the benchmark pattern described in this paper. Additional implementation detail, including workflow steps, prompt definitions, and CSV result structure, is documented in the appendix for readers who want full reproducibility.

3.4 Production Considerations

While these benchmarks focus on controlled, single-accelerator behavior, production deployments introduce additional constraints that influence how these results should be interpreted and applied.

First, real-world workloads are not uniform. Request sizes, output lengths, and traffic patterns vary over time, which means systems must be provisioned for sustained behavior rather than idealized test conditions. The operating ranges identified in this paper should be treated as guidance for steady-state performance, not absolute limits.

Second, production systems must account for burst traffic and tail latency requirements. Running an accelerator at peak observed throughput may maximize efficiency in isolation, but it often leads to unacceptable latency variability under real traffic. In practice, teams should operate below peak concurrency to preserve responsiveness and maintain service-level objectives.

Third, multi-tenant and multi-model environments introduce additional scheduling complexity. While this analysis isolates single-model, single-card behavior, production systems frequently serve multiple models or workloads on shared infrastructure. This can impact both throughput efficiency and latency predictability.

Finally, observability and repeatability are critical. The value of a benchmarking approach like the one presented here is not just in the initial results, but in the ability to re-run tests under different assumptions, validate behavior over time, and adapt to changing workload requirements.

Taken together, these considerations reinforce a key point: production readiness is not determined by peak performance, but by how predictably a system behaves under sustained, variable load.

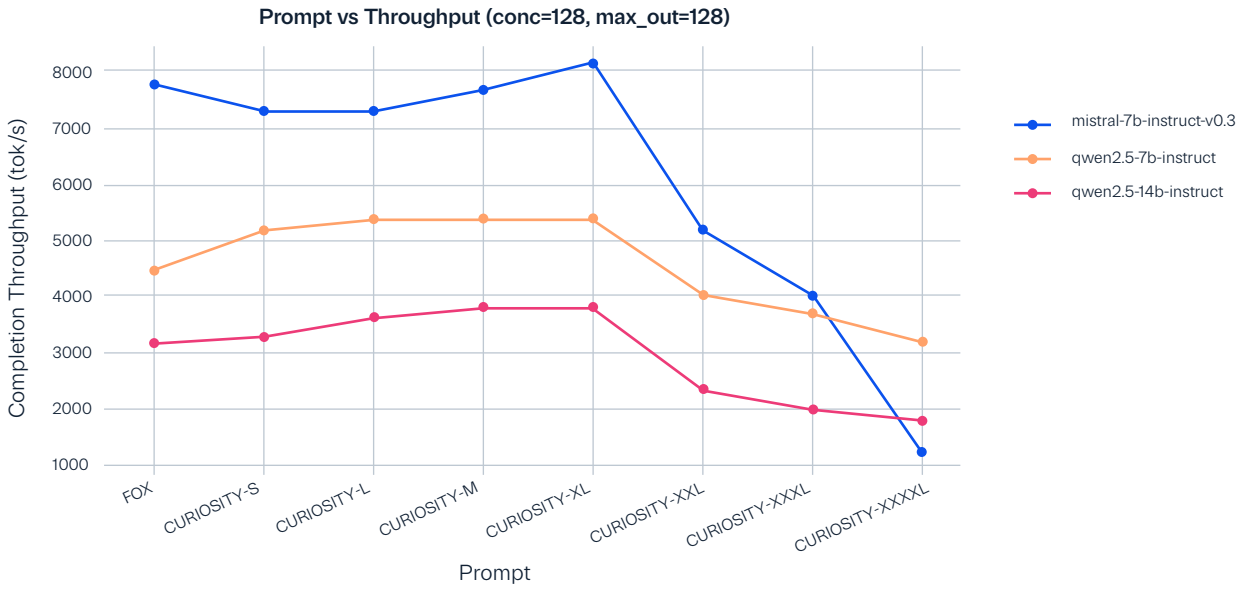
4. Benchmark Analysis

4.1 Throughput: How Many Tokens Can Be Generated

In this section, we examine how throughput on Gaudi 3 changes as prompt size increases across multiple commonly deployed open-weight models.

Prompt Size versus Throughput Behavior

Figure 4.1.1 illustrates throughput as a function of prompt size for all three evaluated models, using a fixed concurrency and output length.



Several patterns emerge immediately.

First, throughput remains relatively stable across small to mid-sized prompts for all models. Excluding the small FOX prompt, throughput remains relatively stable from CURIOSITY-S through CURIOSITY-L. This reflects a regime where the system remains compute-efficient and is not yet dominated by the long-context prefill and KV-cache pressure that appear at the largest prompts. For many real-world inference workloads, this range maps closely to common application behavior, including chat-style interactions, summarization, and lightweight reasoning tasks.

Second, throughput peaks or plateaus before declining as prompt sizes grow further. For the largest prompts in the CURIOSITY-XXL and beyond range, throughput drops sharply. This inflection point is expected and aligns with increased attention computation, KV cache pressure, and longer prefill phases. Importantly, this transition is visible and predictable rather than abrupt or erratic.

Third, model size influences absolute throughput levels, but not the overall shape of the curve. Smaller models such as Mistral 7B sustain higher peak throughput, while larger models like Qwen 2.5 14B operate at lower absolute token rates. However, all models exhibit similar stability across small and mid-sized prompts followed by degradation at very large prompt lengths. This consistency simplifies capacity planning because operating regimes can be reasoned about independently of the specific model choice.

Why This Matters for Inference Workloads

From a deployment perspective, the most important takeaway is not the maximum throughput number, but where throughput remains stable under realistic prompt sizes. Inference services are rarely dominated by extreme prompt lengths. Instead, they tend to operate in a band where requests vary, but cluster around small to mid-sized inputs.

Within this range, Gaudi 3 demonstrates predictable and sustained throughput across all tested models. This makes it well suited for inference-heavy environments where steady-state efficiency matters more than isolated peak measurements. Teams can provision capacity based on expected prompt distributions rather than worst-case extremes, which helps avoid overprovisioning while maintaining performance headroom.

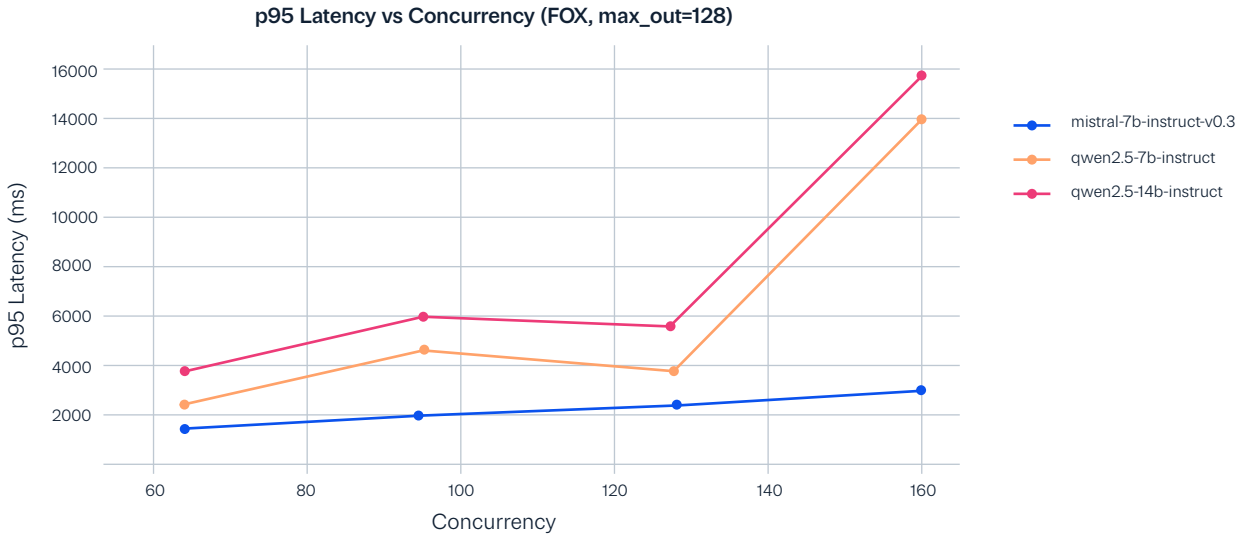
At the same time, the visible throughput decline at very large prompt sizes provides a clear signal about operating boundaries. Rather than obscuring these limits, the benchmarks surface them directly, enabling teams to make informed decisions about prompt constraints, batching strategies, or workload segmentation if long-context requests are expected.

4.2 Latency: How Fast Tokens Are Generated

In this section, we examine how latency on Gaudi 3 evolves as concurrency increases, focusing on both tail latency across models and latency distribution behavior within individual models.

Note: The latency and concurrency sweeps in this section use the FOX prompt (with fixed output length) to isolate concurrency effects under a consistent input profile.

Figure 4.2.1 shows p95 latency as a function of concurrency for all three evaluated models using the FOX prompt and a fixed output length.

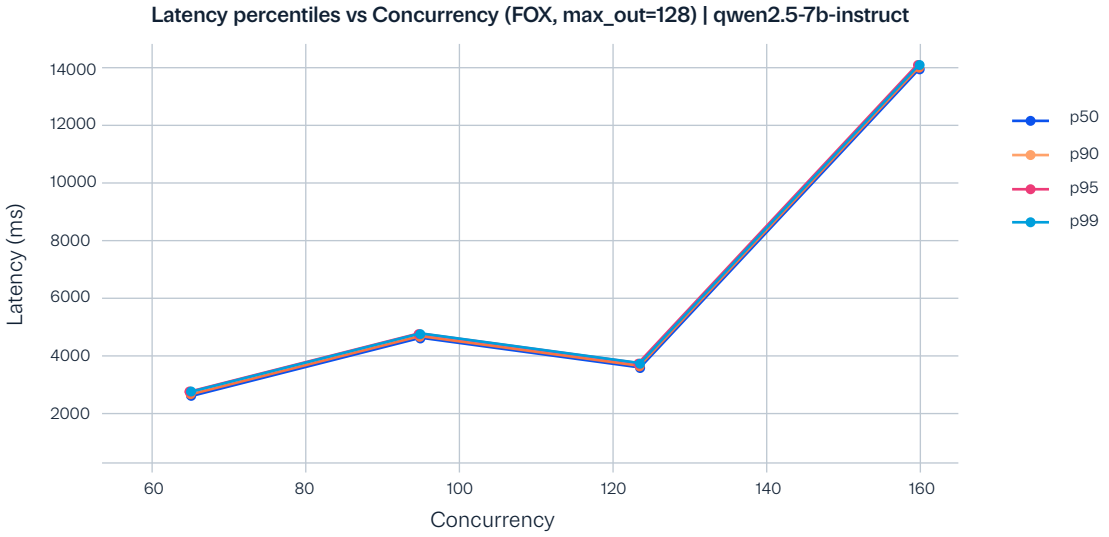


Across all models, latency increases overall as concurrency rises from 64 to 128, with minor non-monotonic variation at intermediate points. Within this range, tail latency growth remains controlled and predictable, indicating that the system can schedule and execute concurrent requests without excessive queueing or contention. This region represents a stable operating band where concurrency can be increased without severely degrading responsiveness.

As concurrency reaches 160, a clear inflection point appears. Tail latency rises sharply for the larger models, particularly Qwen 2.5 7B and 14B, signaling that the accelerator is approaching saturation. This behavior is expected in single-card inference scenarios and reflects increased scheduling pressure and queue buildup rather than instability in the serving stack.

Model size primarily affects absolute latency levels, while saturation effects become more pronounced for larger models at the highest concurrency. Mistral 7B maintains consistently lower tail latency across all concurrency levels, while larger models exhibit higher p95 values. Despite this difference, all models follow a broadly similar scaling pattern, which simplifies reasoning about concurrency limits across model sizes.

Figure 4.2.2 breaks down l**atency percentiles (p50, p90, p95, p99)** for Qwen 2.5 7B as concurrency increases.

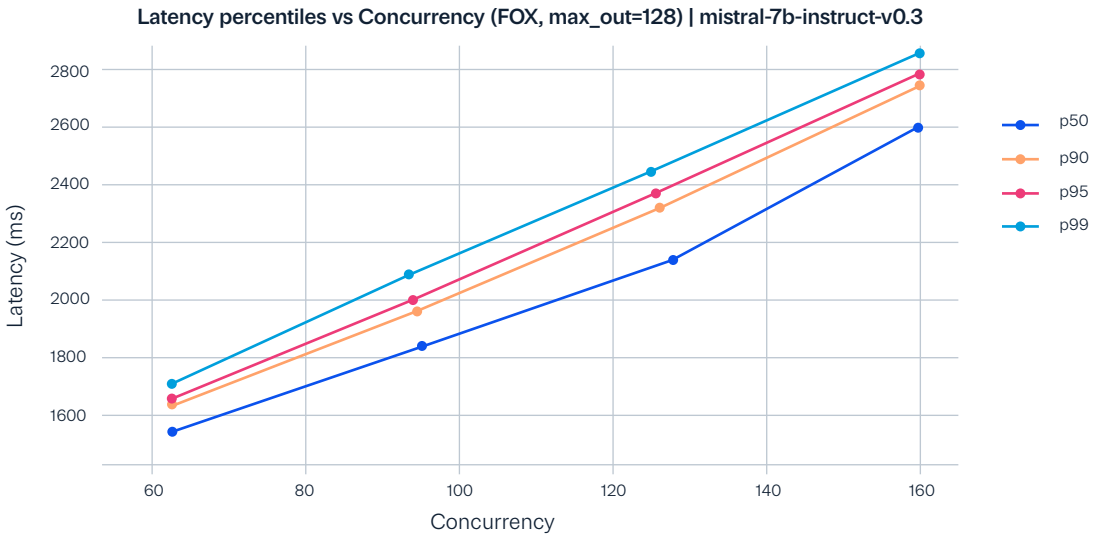


At lower concurrency levels, the latency percentiles remain tightly clustered relative to the overall latency level. Median and tail latencies track closely, indicating consistent request completion times and minimal queueing effects.

As concurrency increases toward 128, latency rises across all percentiles, but the spread remains relatively narrow. This suggests that while requests are taking longer overall, the system is still servicing them in a fairly uniform manner.

At a concurrency of 160, all percentiles increase sharply and remain tightly grouped. This indicates a regime where the system is saturated and nearly all requests experience similarly high latency. Instead of the tail separating dramatically from the median, the entire latency distribution shifts upward, reflecting global resource contention.

Figure 4.2.3 breaks down **latency percentiles (p50, p90, p95, p99)** for Mistral 7B as concurrency increases.



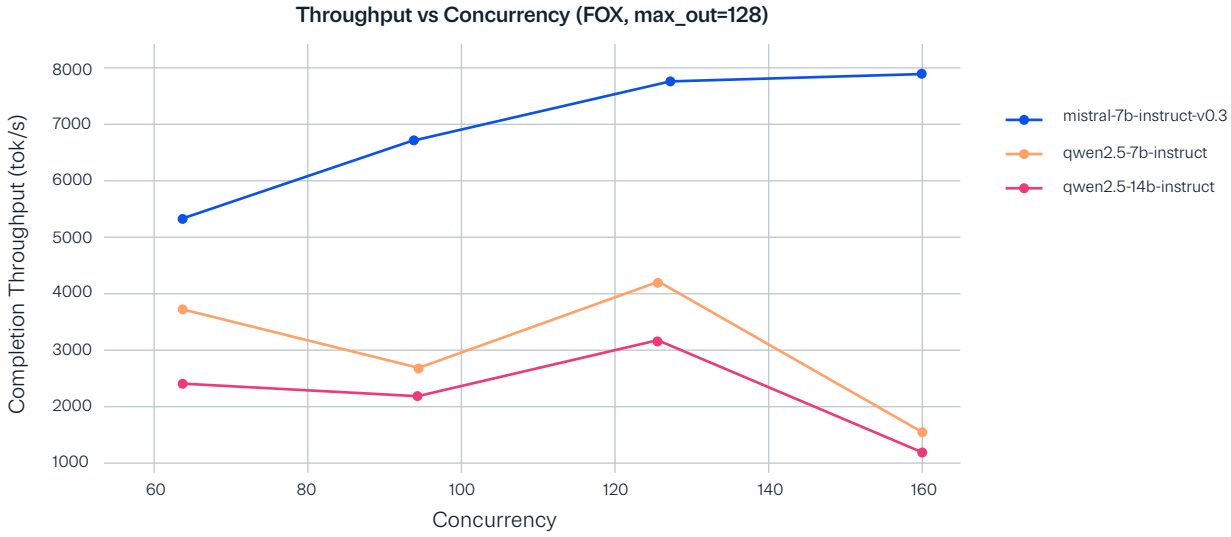
Compared to Qwen 2.5 7B, Mistral 7B exhibits lower absolute latency across all percentiles and a more gradual increase as concurrency rises. The percentile curves remain relatively close compared to Qwen, with a gradual widening as concurrency increases, indicating more predictable latency behavior and less severe tail amplification.

This behavior reflects the lower computational and memory demands of the smaller model. As a result, Mistral 7B can sustain higher concurrency with more predictable latency behavior before entering a saturation regime.

4.3 Concurrency: How Many Requests One Card Can Handle

While throughput and latency describe individual performance characteristics, concurrency determines how those characteristics combine under real traffic. In production inference systems, concurrency defines how many simultaneous requests a single accelerator can sustain before efficiency drops or user experience degrades.

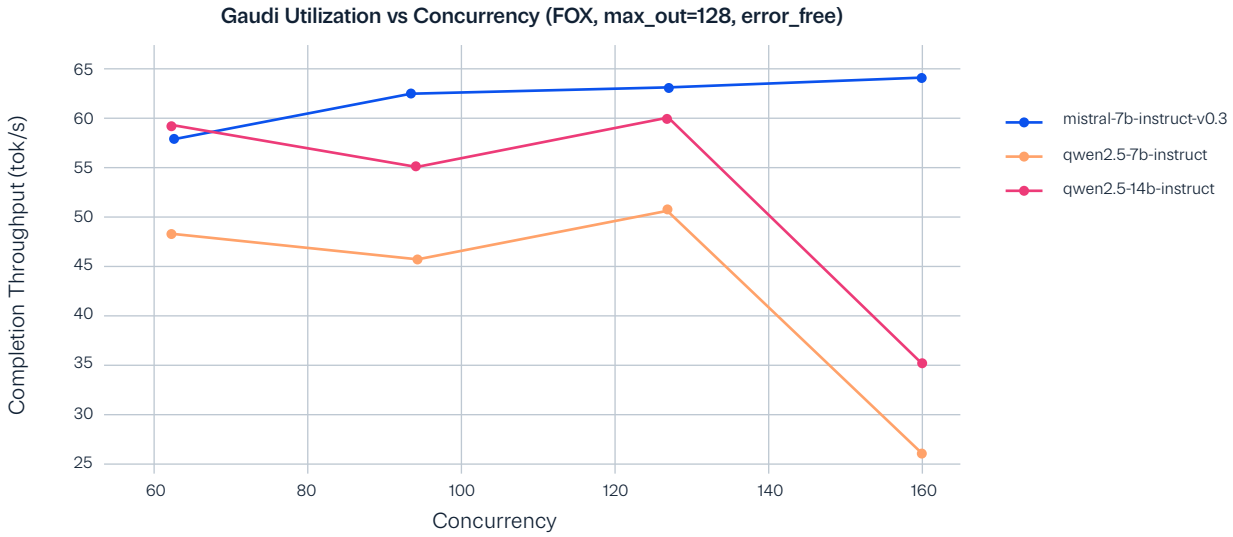
Figure 4.3.1 shows throughput as a function of concurrency for all three models using the FOX prompt with a fixed output length.



As concurrency increases from 64 to 128, throughput rises for all models, indicating that the accelerator is effectively utilizing additional parallelism. This region represents an efficient operating range where increased request volume translates directly into higher total work completed.

Beyond this point, behavior begins to diverge by model size. Mistral 7B continues to scale through a concurrency of 160, with throughput increasing modestly and then flattening. In contrast, both Qwen 2.5 7B and 14B peak near a concurrency of 128 and experience a sharp throughput drop at 160. This collapse indicates that the system has exceeded a practical concurrency limit for larger models, where scheduling pressure and resource contention outweigh the benefits of additional parallel requests.

Figure 4.3.2 provides additional context by showing Gaudi utilization as concurrency increases.



For Mistral 7B, utilization steadily rises and remains high even at the largest concurrency level, aligning with the sustained throughput observed in Figure 4.3.1. For the larger Qwen models, utilization peaks around 128 and then drops significantly at 160, confirming that additional concurrency no longer translates into productive accelerator work.

Taken together, these results highlight an important distinction between theoretical concurrency and effective concurrency. While it is possible to submit more requests to the system, doing so beyond the optimal operating range can reduce overall efficiency and degrade performance, particularly for larger models. For single-card inference deployments, the practical concurrency ceiling appears around 128 concurrent requests for mid-sized models, with smaller models able to sustain higher levels more gracefully.

From a deployment perspective, this matters because concurrency directly affects capacity planning and service stability. Operating near the throughput and utilization peak allows teams to maximize accelerator efficiency while maintaining predictable latency behavior. Pushing beyond that point may increase apparent capacity but often results in lower throughput and worse tail behavior, making it an unattractive trade-off for most production workloads.

4.4 Summary Tables: Practical Operating Ranges for Single-Card Inference

The tables below summarize key operating points observed during the benchmarking runs and are intended to serve as decision-support references rather than exhaustive performance claims. Each table captures representative throughput and latency behavior at selected concurrency levels for a single Gaudi 3 accelerator (single-card inference).

These tables are meant to be a quick reference for readers who want concrete operating points without digging through every chart and sweep.

How to read these tables:

- **Throughput:** average token generation rate at the given concurrency level
- **p50 / p99 latency:** typical vs tail latency behavior at that same operating point
- **“Peak”:** indicates the best observed throughput region for that model in this single-card setup

Model	Concurrency	Throughput (Avg Token/s)	p50 Latency (ms)	p99 Latency (ms)
Mistral 7B	8	~672	1,068	1,163
	64	~4,410	1,500 - 2,600	1,600 - 2,800
	128 (Peak)	~7,500+	2,100 - 2,300	2,400 - 2,700
	256	~5,776	3,981	4,791

Table 4.4.1: Mistral 7B Summary

Model	Concurrency	Throughput (Token/s)	p50 Latency (ms)	p99 Latency (ms)
Qwen 2.5 7B	16	1,362	1,488	1,606
	64	3,793	2,644	2,850
	128 (Peak)	4,253	3,684	4,335
	160	1,481	13,804	14,001

Table 4.4.2: Qwen 2.5 7B Summary

Model	Concurrency	Throughput (Token/s)	p50 Latency (ms)	p99 Latency (ms)
Qwen 2.5 14B	16	787	2,583	2,779
	64	2,314	4,352	4,708
	128 (Peak)	2,748	5,348	6,778
	160	1,322	15,467	15,757

Table 4.4.3: Qwen 2.5 14B Summary

These tables are included as a compact reference for:

- **Selecting practical concurrency limits** per model in a single-card deployment
- **Sanity-checking expected latency ranges** at common operating points
- **Supporting capacity planning discussions** without requiring readers to interpret every plot

They are not intended to replace the deeper chart-by-chart interpretation in Sections 4.1 through 4.3 or the appendix, but to make the outcomes easy to reuse in planning conversations.

5. Next Steps

The benchmarks in this paper are intended to help teams reason about real operating behavior for single-card inference on Gaudi 3, not to claim a universal performance ranking. For many organizations, the next question is not "what is the fastest hardware," but "what is the most practical path to a repeatable, cost-aware inference platform that can be deployed and operated in production."

Based on what we observed, there are a few clear next steps depending on where you are in your inference journey.

5.1 Who Should Care About These Results

This work is most relevant for teams that:

- Expect inference to be a sustained production workload where throughput efficiency and concurrency drive cost.
- Deploy small to mid-sized open-weight models for interactive use cases such as chat assistants, summarization, classification, and retrieval-augmented workflows.
- Need an enterprise-ready approach to serving that emphasizes repeatability, operational clarity, and predictable behavior under load.
- Are evaluating accelerator options and want to understand practical operating bands, not just peak numbers.

5.2 What We Recommend Doing Next

The results presented in this paper are most valuable when applied to a specific workload. To translate these findings into actionable decisions, organizations should take a structured approach to profiling, testing, and deployment planning.

First, profile your inference workload. This includes understanding prompt distribution, output length distribution, target latency (especially p95), sustained QPS requirements, and expected burst behavior. These characteristics directly determine which operating band is appropriate and how sensitive your system will be to concurrency and tail latency.

Second, define a target operating band based on your business requirements. In most enterprise scenarios, maintaining stable p95 latency under load is more important than maximizing peak throughput. Selecting a stable concurrency range where latency remains predictable will result in more reliable production performance.

Third, validate results using your own data and constraints. Reproduce the benchmark pattern using your actual prompts, output lengths, and any safety or post-processing logic. This ensures that observed performance aligns with real-world usage rather than synthetic or simplified workloads.

Finally, determine an appropriate silicon placement strategy across your AI platform. Different workloads may be better suited to different compute profiles. Training and certain software ecosystems may align with GPU-based environments, while Gaudi-based systems can provide strong price-performance for sustained inference workloads. CPU-based deployments may be appropriate for lightweight models, embeddings, or low-throughput services.

Taken together, these steps provide a practical path from benchmark results to production-ready architecture decisions.

5.3 How We Can Help

Translating benchmark results into a production-ready AI platform requires more than raw performance data. It requires alignment between workload characteristics, infrastructure design, and operational practices.

We support organizations across this lifecycle, from initial assessment through full-scale deployment.

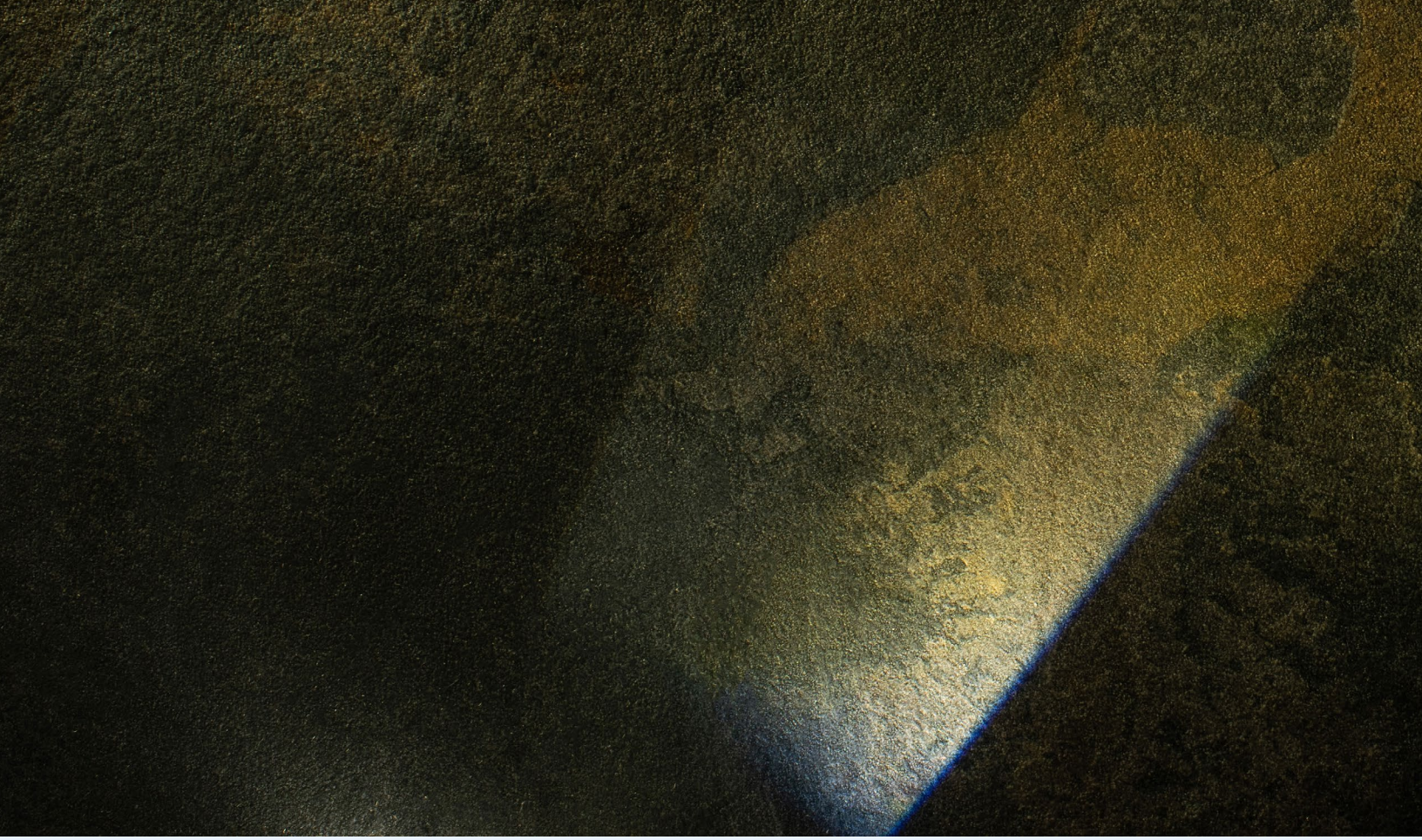
Our inference readiness workshops help teams profile their workloads and build a clear understanding of unit economics. This includes mapping prompt and output characteristics to throughput, latency targets, and infrastructure requirements.

We provide OpenShift AI reference architectures designed for enterprise environments. These include multi-tenant configurations, resource quotas, observability frameworks, RBAC models, and CI/CD integration to support scalable and governed AI operations.

For organizations evaluating hardware platforms, we offer benchmark replication engagements. These allow customers to run their own models across Gaudi and GPU-based environments using a consistent methodology, enabling direct comparison under realistic conditions.

Finally, we support production rollout with proven SRE patterns, including monitoring dashboards, autoscaling strategies, and incident response playbooks. The focus is not only on achieving performance targets, but on maintaining them reliably under real-world conditions.

Our goal is to help organizations move from experimentation to predictable, production-grade AI systems.



Author:

Matthew Adkins, Technical Consultant, AI & Cloud Solutions

Matthew is a Technical Consultant at AHEAD, specializing in data science and enterprise AI solutions. With an educational background in Electrical Engineering and Economics, Matthew brings a unique business-grounded perspective to the AI solutions he designs, develops, and operates. Having deep experience building and utilizing on-premise AI solutions, Matthew provides meaningful partnerships and guidance to enterprises along their AI journey.

AHEAD

Combining cloud-native capabilities in software and data engineering with an unparalleled track record of modernizing infrastructure, we're uniquely positioned to help accelerate the promise of digital transformation.

Visit us at ahead.com.

National Hubs

CHICAGO
444 W. Lake Street
Suite 3000
Chicago, IL 60606

NEW YORK
500 5th Avenue
Floor 17
New York, NY 10010

ATLANTA
1117 Perimeter Center
W406
Atlanta, GA 30338

SAN FRANCISCO
2000 Crow Canyon Place
Suite 250
San Ramon, CA 94583

Appendix

Appendix A. Benchmark Scope and Reproducibility Overview

This appendix provides the full technical context required to reproduce the benchmarks presented in this paper. It documents the exact workflow, environment assumptions, tooling, prompt definitions, and metric collection mechanisms used during testing.

The intent of this appendix is transparency and reproducibility, not additional analysis. All interpretation of results is contained in the main body of the paper. Readers seeking to understand or replicate the benchmark mechanics can rely on the material below as a complete reference.

All benchmarks were executed using a fixed, notebook-driven workflow in a Red Hat OpenShift AI environment with Intel Gaudi 3 accelerators, KServe serverless model serving, and Service Mesh enabled.

Appendix B. Notebook-Driven Benchmarking Workflow

All benchmark runs were executed using exactly four notebooks, with no manual YAML editing or ad hoc scripts. All runtime and InferenceService manifests were generated programmatically inside the notebooks to ensure consistency.

Notebook set

1. 01_CREATE_vLLM_GAUDI_SERVINGRUNTIME.ipynb
2. 02_DOWNLOAD_MODEL_WEIGHTS.ipynb
3. 03_CREATE_INFERENCESERVICE_YAML.ipynb
4. 04_RUN_BENCHMARK.ipynb

Model weights and all benchmark artifacts were stored on persistent volumes and were not bundled with the notebooks.

Appendix C. High-Level Execution Flow

Each benchmark followed a consistent, notebook-driven execution flow designed to minimize manual intervention and ensure repeatability across models and parameter sweeps. Rather than treating every run as a full environment rebuild, the workflow distinguishes between one-time environment preparation steps and per-model or per-run execution steps.

At a high level, the benchmarking process consists of three phases: environment preparation, model deployment, and benchmark execution.

Environment Preparation (Performed Once per Project)

A dedicated OpenShift AI Data Science Project was created to isolate resources and ensure consistent configuration. Two persistent volume claims were provisioned and attached to a single Jupyter workbench: one for notebooks and workspace state, and one for model weights and benchmark artifacts.

The workbench environment was then validated to ensure:

- Gaudi devices were visible and schedulable
- The project namespace was registered with the Service Mesh
- KServe serverless components were available and ready

These checks are prerequisites for successful model serving and were not repeated for every benchmark run once the environment was confirmed healthy.

Model Deployment (Performed Once per Model)

For each model evaluated, weights were downloaded once to the shared model PVC. A Gaudi-enabled vLLM ServingRuntime was generated and applied using a notebook-driven process, followed by generation and application of a model-specific InferenceService.

All ServingRuntime and InferenceService manifests were created programmatically inside the notebooks. No manual YAML editing was performed. Each model was served using a single-model-per-pod configuration to isolate performance behavior.

Once the InferenceService reached a Ready state, the model endpoint was reused for all subsequent parameter sweeps involving that model.

Benchmark Execution and Result Collection (Performed Per Run)

Benchmark execution was performed using a single notebook that drove all runtime variation. For each run, the notebook:

- Issued OpenAI-style completion requests to the vLLM endpoint
- Varied prompt size, output length, and concurrency according to a predefined checklist
- Measured throughput and latency directly from the vLLM API
- Queried Gaudi utilization metrics from Prometheus via the Thanos querier
- Parsed and validated all collected metrics

Each completed run appended a structured row to a cumulative CSV file stored on the model PVC. This CSV serves as the authoritative source for all plots, tables, and summaries presented in the main body of the post.

This execution model allowed consistent reuse of the same environment, runtime, and model deployment while varying only the parameters under study. As a result, observed performance differences can be attributed to prompt characteristics, concurrency, and model size rather than environmental drift or configuration inconsistency.

Appendix D. OpenShift AI Environment Assumptions

Hardware

- Intel Gaudi 3 accelerators available on worker nodes
- Single-accelerator (single-card) inference per benchmark run
- Adequate system memory and storage for model weights and KV cache

Software and Platform

- Red Hat OpenShift with OpenShift AI installed
- KServe serverless enabled

- Service Mesh enabled and configured
- Gaudi software stack aligned with firmware and drivers
- Gaudi-optimized vLLM container image
- Prometheus monitoring stack with Thanos querier

All results presented in this post reflect single-card inference behavior only.

Appendix E. Persistent Storage Configuration

Two persistent volume claims were required:

Workbench PVC

- Purpose: notebooks and workspace
- Mount path: /opt/app-root/src
- Mode: ReadWriteOnce
- Size: 20Gi

Model and Results PVC

- Purpose: model weights, logs, benchmark outputs
- Mount path: /mnt/models
- Mode: ReadWriteOnce
- Size: 300Gi

PVC health was verified inside the running workbench pod by writing test files to both mount points.

Appendix F. Known OpenShift AI and SCC Considerations

Several environment-level issues are common in enterprise or partner lab environments:

- Notebook UID falling outside the allowed SCC range
- Invalid secure file modes on NFS-mounted Jupyter runtime directories
- Duplicate NOTEBOOK_ARGS injected by the platform
- Missing Service Mesh membership preventing KServe serverless readiness

In this environment, these issues were resolved by:

- Setting fsGroup and runAsUser to an allowed UID range
- Relocating the Jupyter runtime directory to /tmp/jupyter-runtime
- Using a minimal, non-conflicting NOTEBOOK_ARGS configuration
- Explicitly adding the project namespace to the ServiceMeshMemberRoll

Once resolved, the environment remained stable across all benchmark runs.

Appendix G. Model Serving Configuration

All models were served using:

- A Gaudi-enabled vLLM ServingRuntime
- Single-model-per-pod configuration
- PVC-based model loading
- OpenAI-compatible vLLM API endpoints

Key runtime characteristics:

- Gaudi devices exposed via habana.ai/gaudi
- vLLM configured for HPU execution
- HPU Graph warmup enabled
- Consistent resource configuration across models

Warmup phases were observed for all models and are expected behavior, particularly for larger models.

Appendix H. Prompt Set and Token Scaling

All benchmarks were executed using a fixed, deterministic prompt set designed to scale input length in a controlled and interpretable way. The purpose of this prompt design was to isolate the impact of input token count on throughput, latency, and concurrency behavior, while avoiding variability introduced by changes in task structure, tone, or instruction complexity.

The same prompt text was reused across all models. Only prompt length, maximum output tokens, and concurrency were varied during benchmarking. No prompts were generated dynamically, randomized, or modified between runs.

Prompt Families

Two primary prompt families were used throughout the benchmark runs.

The first is a short, fixed prompt used to isolate concurrency effects under a minimal and stable input profile. The second is a progressively expanding prompt series designed to increase input token count while preserving semantic continuity and instruction structure.

This approach ensures that observed performance differences are driven primarily by token volume rather than changes in task type or prompt style.

FOX Prompt

The FOX prompt is a short, static input used exclusively for concurrency and output-length sweeps.

Prompt text:

The quick brown fox jumps over the lazy dog.

Because of its minimal length and lack of structural complexity, this prompt minimizes prefill overhead and attention computation. This allows concurrency effects to dominate observed behavior, making it well suited for identifying saturation points, throughput scaling, and latency inflection under increasing request volume.

CURIOSITY Prompt Series

The CURIOSITY prompt series is designed to progressively increase input length while maintaining a consistent topic, narrative flow, and instruction style. Each prompt expands on the previous one by adding

additional explanatory detail, rather than introducing new tasks or formats.

This design allows prompt size to scale in a controlled manner while preserving comparability across runs.

CURIOSITY-S

Prompt text:

Explain the importance of curiosity.

CURIOSITY-M

Prompt text:

Explain why curiosity is important for learning, problem solving, creativity, and adapting to new situations in everyday life.

CURIOSITY-L

Prompt text:

Explain why curiosity is important for learning and personal growth, including how asking questions, exploring ideas, and seeking understanding helps people solve problems, adapt to change, develop creativity, and make better decisions in both professional and everyday situations.

CURIOSITY-XL

Prompt text:

Explain why curiosity is important for learning, personal growth, and problem solving. Discuss how curiosity encourages people to ask questions, explore unfamiliar ideas, challenge assumptions, and seek deeper understanding. Include how curiosity supports creativity, adaptability, and continuous improvement in both professional environments and everyday life, helping individuals make better decisions, learn new skills, and respond effectively to change.

CURIOSITY-XXL

Prompt text:

Explain why curiosity is important for learning, personal growth, and effective problem solving. Begin by describing curiosity as a natural desire to understand how things work, why events occur, and how ideas connect. Explain how curiosity drives people to ask questions, seek new information, and explore unfamiliar topics rather than relying only on existing knowledge. Discuss how this mindset supports deeper learning by encouraging individuals to go beyond memorization and develop true understanding. Describe how curiosity contributes to creativity by promoting experimentation, open minded thinking, and the willingness to consider alternative perspectives. Explain how curious individuals are more likely to generate new ideas, identify unexpected connections, and approach challenges with flexibility. Discuss the role of curiosity in adaptability, particularly in environments that are constantly changing, where learning new skills and updating knowledge is essential. Include examples of how curiosity benefits professional settings, such as improving problem solving, supporting innovation, and enhancing collaboration. Explain how curious employees are more likely to ask clarifying questions, learn from feedback, and continuously improve their performance. Also discuss how curiosity helps leaders make better decisions by encouraging them to seek diverse viewpoints and understand complex situations more fully. Finally, explain how curiosity enriches everyday life by helping people understand others, develop new interests, and navigate uncertainty with confidence. Conclude by emphasizing why curiosity is a critical trait for lifelong learning, resilience, and personal development in a rapidly evolving world.

CURIOSITY-XXXL

Prompt text:

Explain why curiosity is one of the most important drivers of learning, personal growth, creativity, and effective problem solving across both professional and everyday contexts. Begin by defining curiosity as a fundamental human desire to understand how things work, why events occur, and how ideas, systems, and experiences are connected. Describe how curiosity motivates people to ask questions, seek explanations, and explore unfamiliar topics rather than relying solely on prior knowledge or assumptions. Discuss how curiosity plays a critical role in learning by encouraging individuals to move beyond surface level memorization and toward deeper understanding. Explain how asking “why” and “how” helps learners identify patterns, build mental models, and retain knowledge more effectively. Describe how curiosity supports lifelong learning by motivating people to continuously update their skills and understanding as new information becomes available. Explain the relationship between curiosity and problem solving. Describe how curious individuals are more likely to examine problems from multiple angles, question initial assumptions, and experiment with alternative solutions. Discuss how curiosity enables people to break complex problems into smaller components, investigate root causes, and identify unexpected connections that lead to more effective solutions. Include examples of how curiosity helps individuals respond constructively when initial solutions fail.

CURIOSITY-XXXXL

Prompt text:

Explain why curiosity is one of the most important drivers of learning, personal growth, creativity, and effective problem solving across both professional and everyday contexts. Begin by defining curiosity as a fundamental human desire to understand how things work, why events occur, and how ideas, systems, and experiences are connected. Describe how curiosity motivates people to ask questions, seek explanations, and explore unfamiliar topics rather than relying solely on prior knowledge or assumptions. Discuss how curiosity plays a critical role in learning by encouraging individuals to move beyond surface level memorization and toward deeper understanding. Explain how asking “why” and “how” helps learners identify patterns, build mental models, and retain knowledge more effectively. Describe how curiosity supports lifelong learning by motivating people to continuously update their skills and understanding as new information becomes available. Explain the relationship between curiosity and problem solving. Describe how curious individuals are more likely to examine problems from multiple angles, question initial assumptions, and experiment with alternative solutions. Discuss how curiosity enables people to break complex problems into smaller components, investigate root causes, and identify unexpected connections that lead to more effective solutions. Include examples of how curiosity helps individuals respond constructively when initial solutions fail. Explore how curiosity contributes to creativity by encouraging experimentation, open mindedness, and the willingness to explore unconventional ideas. Discuss how curiosity supports adaptability in rapidly changing environments, where continuous learning and skill development are essential. Include discussion of curiosity in professional settings, such as its impact on innovation, collaboration, and leadership decision making. Conclude by explaining how curiosity enriches everyday life by supporting empathy, self reflection, and openness to new experiences.

Prompt Token Accounting

Prompt token counts were measured automatically by the benchmarking pipeline during each run. Token counts were recorded per request and stored alongside throughput and latency metrics in the cumulative results summary CSV. No manual token estimation or post hoc adjustment was performed.

Prompt text, prompt length category, and output length parameters are included in the results metadata to ensure traceability between reported metrics and the exact input used during each benchmark run.

Appendix I. Benchmark Parameter Coverage

Benchmarks swept across combinations of:

- Concurrency
- Prompt size
- Maximum output tokens

Representative coverage included:

- Concurrency values ranging from low tens to saturation regimes
- Output lengths from short completions to long responses
- Prompt sizes spanning multiple orders of magnitude

Each model followed a structured checklist to ensure comparable coverage across runs.

Appendix J. Metric Collection and Definitions

Each benchmark run collected the following metrics:

- Tokens per second (completion and total)
- p50, p90, p95, and p99 latency
- Minimum and maximum latency
- Successful and failed request counts
- Error rate
- Gaudi utilization percentage

Latency metrics were measured at the vLLM API level using OpenAI-style completion requests.

Appendix K. Gaudi Utilization Measurement

Gaudi utilization was captured using Prometheus rather than node-level tools.

- Metrics queried from the in-cluster Thanos querier
- Metric used: habanalabs_utilization
- Time-windowed average during the benchmark run
- Authentication via service account token
- TLS verification handled automatically with fallback logging

If utilization capture failed due to monitoring or RBAC configuration, the benchmark still completed successfully and recorded utilization as unavailable without impacting throughput or latency measurements.

Appendix L. Results Aggregation and CSV Schema

All benchmark outputs were aggregated into a single cumulative CSV file stored on the model and results persistent volume. This file functions as the authoritative, append-only record for every benchmark run executed during the study.

Each row in the CSV represents one complete benchmark execution. Rows are written once per run and never modified or overwritten.

At a high level, each row captures the following categories of information:

- Model identity and serving configuration
- Prompt metadata and prompt length classification
- Concurrency and output length parameters
- Throughput measurements
- Latency distribution metrics
- Request success and error counts
- Gaudi utilization metrics and measurement window
- Execution timestamp

To make this structure concrete, the table below illustrates a representative example row with simplified values. This is provided for clarity only; exact values vary by model and run.

Example CSV row (simplified):

Field	Example Value
model_name	mistral-7b-instruct-v0.3
prompt_family	FOX
prompt_size_label	FOX
max_output_tokens	128
concurrency	128
tokens_per_second	7420.3
latency_p50_ms	2185
latency_p95_ms	2450
latency_p99_ms	2690
requests_successful	128
requests_failed	0
gaudi_util_avg_percent	91.4

Field	Example Value
gaudi_util_window_start	2025-12-26T17:14:02Z
gaudi_util_window_end	2025-12-26T17:15:12Z
run_timestamp	2025-12-26T17:15:30Z

The full CSV includes additional supporting fields such as p90 latency, minimum and maximum latency, error rate, and internal identifiers used by the benchmark harness. However, the fields shown above are sufficient to understand how all tables and figures in the main post are derived.

All summary tables, plots, and interpretations presented in the main body of the post were generated exclusively from this CSV. No manual aggregation, filtering, or post-processing outside the recorded results was performed.

Appendix M. Reuse and Extension Guidance

To benchmark additional models or configurations:

- Download model weights to the model PVC
- Generate a new InferenceService using Notebook 03
- Run the benchmark using Notebook 04
- Append results automatically to the summary CSV

The same workflow can be extended to:

- Multi-card inference
- Alternative batching strategies
- Different traffic profiles
- Additional model families

Appendix N. Summary

This appendix exists to make the results in this post verifiable, reproducible, and reusable. No claims in the main body rely on undocumented configuration, hidden scripts, or implicit assumptions. All results can be reproduced by following the documented workflow using the same platform components and parameter ranges.