POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Benchmarking Cloud Instances for Generative AI Inference: The Impact of Accelerators and Serving Hyperparameters on Cost-Performance Frontiers

Tesi di Laurea Magistrale in
Computer Science and Engineering -
Ingegneria Informatica

Author: **Niccolò Anesa**

Student ID: 259961
Advisor: Prof. Paolo Cremonesi
Co-advisors: Stefano Cereda
Academic Year: 2025-26

Abstract

Deploying Large Language Models at scale is expensive and technically demanding, making inference optimization a critical priority for any organization self-hosting AI workloads. This thesis benchmarks two cloud hardware accelerators — the NVIDIA T4 GPU (g4dn.xlarge) and the AWS Inferentia2 ASIC (inf2.xlarge) — across a controlled grid of 198 configurations combining two open-source transformer models (Meta's Llama and Alibaba's Qwen) with a systematic sweep of vLLM inference engine hyperparameters. Each configuration was evaluated under synthetic load generated by GuideLLM, collecting high-resolution telemetry on throughput, Time to First Token, Time Per Output Token, KV cache utilization and infrastructure cost per million output tokens. The experimental pipeline was designed for reproducibility and generality, pairing an automated benchmark orchestration script with a structured telemetry collection layer that produced a unified dataset for the subsequent analysis. Performance results were interpreted through Pareto efficiency frontiers mapping each metric against cost, enabling both hardware comparison and optimal hyperparameter identification across different deployment objectives: throughput maximization, latency minimization and cost reduction. The GPU shows a modest cost and performance edge over the ASIC, which proves ideal in scenarios where workload predictability and resource stability outweigh the pursuit of peak efficiency. At the optimal frontier, the GPU achieves a 2.37x lower cost per million tokens. At their respective optimal configurations, this advantage extends to raw performance: 64% higher throughput, 96% lower Time to First Token and 56% lower inter-token latency than the Inferentia2 ASIC. The analysis also validated the impact of individual hyperparameters on inference performance and identified the optimal configurations for each specific use case, with the request concurrency level being the main driver for performance tuning along the optimal curves. This research was conducted in collaboration with Moviri, an IT consulting company specializing in performance engineering, analytics and cybersecurity.

Keywords: LLM inference, cloud benchmarking, hardware accelerators, vLLM, cost optimization

Abstract in lingua italiana

Il deployment su larga scala di Large Language Models è costoso e tecnicamente complesso, rendendo l'ottimizzazione dell'inferenza una priorità per qualsiasi organizzazione che gestisca internamente carichi di lavoro AI. Questa tesi confronta due acceleratori hardware cloud — la GPU NVIDIA T4 (g4dn.xlarge) e l'ASIC AWS Inferentia2 (inf2.xlarge) — attraverso una griglia controllata di 198 configurazioni che combinano due modelli transformer open-source (Llama di Meta e Qwen di Alibaba) con una ricerca sistematica sugli iperparametri del motore di inferenza vLLM. Ogni configurazione è stata valutata sotto carico sintetico generato da GuideLLM, raccogliendo telemetria ad alta risoluzione su throughput, Time to First Token, Time Per Output Token, utilizzo della KV cache e costo infrastrutturale per milione di token generati. La pipeline sperimentale è stata progettata per garantire riproducibilità e generalità, abbinando uno script di orchestrazione automatizzata dei benchmark a uno strato strutturato di raccolta telemetrica che ha prodotto un dataset unificato per l'analisi successiva. I risultati di performance sono stati interpretati attraverso frontiere di efficienza di Pareto che mettono in relazione ciascuna metrica con il costo, consentendo sia il confronto tra architetture che l'identificazione della configurazione ottimale per diversi obiettivi di deployment: massimizzazione del throughput, minimizzazione della latenza e riduzione dei costi. La GPU mostra un modesto vantaggio in termini di costo e performance rispetto all'ASIC, che risulta ideale negli scenari in cui la prevedibilità del carico di lavoro e la stabilità delle risorse hanno la priorità rispetto alla ricerca dell'efficienza massima. Sulla frontiera ottimale, la GPU raggiunge un costo per milione di token 2,37 volte inferiore. Nelle rispettive configurazioni ottimali, questo vantaggio si estende alle metriche di performance: throughput superiore del 64%, Time to First Token inferiore del 96% e latenza inter-token inferiore del 56% rispetto all'ASIC Inferentia2. L'analisi ha inoltre validato l'impatto dei singoli iperparametri sulle performance di inferenza e identificato le configurazioni ottimali per ciascun caso d'uso specifico, individuando nel livello di concorrenza delle richieste il principale contributore all'ottimizzazione delle prestazioni lungo le curve di ottimo. La ricerca è stata condotta in collaborazione con Moviri, società di consulenza IT specializzata in performance engineering, analytics e cybersecurity.

Parole chiave: inferenza LLM, benchmarking cloud, acceleratori hardware, vLLM, ottimizzazione costi

Contents

Abstract **i**

Abstract in lingua italiana **iii**

Contents **v**

1 Introduction **1**
1.1 The Generative AI Bottleneck 2
1.2 Research Objectives 4
1.3 Thesis Structure 5

2 Theoretical Background **7**
2.1 LLM Generation Architecture 8
2.2 The Hardware Landscape: GPUs vs. ASICs 10
2.3 The Role of Inference Engines 12
2.4 Inference Software Innovations 15
2.5 Hardware Execution Paradigms 16
2.5.1 GPU's Dynamic Execution 16
2.5.2 AWS Neuron's Static AOT Compilation 17
2.6 Open-Source LLM Architectures 18
2.6.1 Serving Infrastructure Control 19
2.6.2 Training Paradigms 20
2.6.3 Model Dimensions 20
2.6.4 Prompt and Context Processing 21

3 Experimental Methodology **23**
3.1 Infrastructure and Test Environments 24
3.1.1 Cloud Instance Types 24
3.1.2 The Machine Image 27

3.2 vLLM Configuration 28
3.2.1 Installation 28
3.2.2 Model Serving 29
3.2.3 Model compilation on Neuron 30
3.3 The Load Generator 32
3.4 Experimental Pipeline 34
3.4.1 The Iterative Process 34
3.4.2 The Hyperparameter Search Grid 34
3.5 Telemetry Infrastructure 36
3.6 Dataset Description 38

4 Empirical Performance and System Boundaries 41
4.1 Run Reliability and System Failures 41
4.2 The Efficient Frontier 43
4.3 Hyperparameter Interaction 45
4.4 Hardware Comparison 46
4.5 Financial Efficiency Analysis 49

5 Conclusions and Future Developments 53
5.1 Summary of Contributions 53
5.2 Recommendations 53
5.3 Limitations of the Study 54
5.4 Future Work 55

Bibliography 57

A Appendix: benchmark execution scripts 63

List of Figures 65

List of Tables 67

1 | Introduction

Despite the “intelligence” inherent in their name, large language models are far from plug-and-play when it comes to deployment. Model serving for tech companies — but also non tech ones — is as hard as delivering a standard distributed application to production stage, or sometimes harder and more expensive, due to their immense demand for infrastructural resources. Coupled with the efficient but complicated management of cloud resources, costs can bloat and performance gain cannot always keep pace. Lots of academic and professional material exists for optimal training of Large Language Models, surely more than the amount of literature items existing regarding inference, even though this represents the biggest portion of AI expenses [28], especially considering that training is done once for each new model version, and most of the companies don’t need and aren’t training their own models, often they don’t even fine-tune them [29]. Jensen Huang, the CEO of the most profitable company of the last years — NVIDIA — at the GTC 2026 keynote speaks about data centers becoming "AI Factories" whose product are tokens, and as any factory, the more it is produced, the more the revenue, highlighting the need for more powerful and reliable model serving. He also acknowledges inference as today’s biggest challenge in AI, being both difficult and critical as it drives revenue [19].

Taking this into account, optimization of AI model serving should be considered a top priority for both technical and FinOps reasons. Beyond the existing urgency surrounding this topic, it is also worth noting that recently the cost of hardware increased due to the scarcity of raw materials compared to the increasing demand for general digitalization and AI-driven industrialization [39].

Not to be ignored is the possibility for companies to just purchase *Model as a Service*/*Inference as a Service* and delegate all management to the provider, though this choice usually is not the cheapest and in some cases might not be even suitable when data regulation and complete customization of the full stack is the priority [27].

This thesis will focus on the first option of the make or buy dilemma, trying to get the most out of the choice of in-house serving strategy. The research was conducted in collaboration with Moviri, a multinational IT consulting company with headquarters

in Milan, specializing in performance engineering, analytics and cybersecurity, whose operational experience with enterprise AI workloads informed the experimental design and analysis. The empirical results show that the GPU holds a modest edge over the Inferentia2 ASIC: at the optimal frontier it achieves a 2.37x lower cost per million tokens, while at their respective optimal configurations the performance gap reaches 64

Training is all about raw power and doing the most in the shortest amount of time, inference has to shift focus on business continuity, availability, reliability and fault tolerance. The timespan is potentially infinite so the variable of time is to be considered in a continuous manner, and in cloud infrastructure billing it is placed at the denominator, as independent variable, making it a value not to be reduced but an inevitable multiplier for raw price tags, which are the ones to be reduced [38]. As standard enterprise practice nowadays, LLM inference for these experiments have been run on cloud instances, related costs and scaling considerations will be made to contribute to the analyses' result generality. In particular, Amazon Web Services EC2, one of industry leaders in cloud computing, has been selected to host the model serving and the selection of different hardware types and sizes, made the benchmarking possible between dedicated ASICs and general purpose GPUs.

1.1. The Generative AI Bottleneck

Large Language Models (LLMs) are massive deep learning networks, primarily built upon the Transformer architecture, pre-trained on vast datasets to comprehend and probabilistically model human language. In the context of system deployment, "inference" refers to the operational phase where these pre-trained models are actively queried to generate responses. During this phase, the model ingests a user's prompt to understand the context, and then autoregressively calculates the mathematical probability of the next token, generating the output step-by-step in a continuous sequence of tokens.

Corporations are increasingly driven to self-host LLMs to automate enterprise workflows and to serve LLM-backed chatbots for internal use or as a product for customers. The growing demand for rapid and financially viable model deployments motivates ongoing research into infrastructure optimization. When hosting LLMs and Chatbots, having already tackled model response quality, optimizing inference means working mainly on three KPIs: latency, throughput and infrastructure costs. Here in detail:

- **Latency**: The time delay between sending a prompt (input) to the model and receiving the corresponding response (output). A faster response is the driver of the perceived quality of any application, from the user perspective it means more

interaction with the system and less time to achieve its goal. From the company perspective, this translates into defining Service Level Agreements to guarantee a satisfying user experience. In practice, the metrics that represent LLM latency are: Time to First Token (TTFT); Time Per Output Token (TPOT) or Inter-token Latency (ITL)

- **Throughput**: The volume of data — specifically tokens — that an AI system processes, generates, or transmits within a specific unit of time. More answers produced in a certain amount of time mean more users can be connected at the same time and, in general, more requests can be managed. In practice, the metrics that represent LLM throughput are: Tokens per Second (TPS); Requests per Minute (RPM).
- **Infrastructure costs**: Unoptimized resource consumption leads to more computational infrastructure demand which, not surprisingly, implies paying more, and not always the computational power is proportional to the costs. This thesis covers cloud infrastructure costs in depth, which are usually in €/h, but similar considerations can be made for on-premise infrastructure costs, for which an initial investment is amortized, or for MaaS costs, billed pay-as-you-go per million token or with fixed price subscription with cost proportional to a utilization limit.

Throughput can be improved as a direct consequence of an increase in parallel computation, thus increasing the latency for the single call with fixed computational power which implies that the sweet spot to be found is the right trade-off between the two metrics. The optimization of these metrics usually leans towards improving one or two of them by fixing the other(s) according to the business needs and capabilities. When the primary objective is to increase application capacity, the straightforward approach is to scale up or scale out on hardware, but sooner or later the expenses or hardware availability itself will hit a threshold for which installing more is inconvenient and becomes harmful for the business, so other strategies are necessary to accommodate the increasing performance demand. To tackle an operational research problem means in almost all cases to set fixed variables and find the best values for the others: fixing costs forces the improvement of model performance by means other than increasing raw power; fixing also one of the performance metrics (e.g. latency) helps to isolate the problem on an even finer level (e.g. throughput increase objective).

Elsewhere, performance might be the first fixed variable, for example when Service Level Agreements are set in a company or when the demand is stable and does not require upgrades. This comes down to the investigation of pure cost reduction and saving challenges.

Furthermore, uncommon domain-specific or business-specific needs can dictate the direction to which optimization and/or upgrades are set: an hypothetical emergency chatbot needs to provide fast and accurate responses, this is a latency optimization problem; an AI assistant for shopping during flash sales should account primarily for high throughput as large amount of users could access the platform at the same time; an AI-driven innovative startup could need advanced models but keep an eye on minimal cost variance.

In every deployment scenario, finding the exact optimal operating conditions makes the difference between commercial success and prohibitive infrastructure overhead. Last but not least, companies could want — and should — optimize the hardware load generated by AI inference as a way to reduce environmental impact, a critical topic in modern large-scale computation, especially in AI systems [46].

1.2. Research Objectives

Benchmarking is defined by the Cambridge Dictionary [12] as *the act of measuring the quality of something by comparing it with something else of an accepted standard*. This thesis will explore the differences in AI acceleration hardware. The "golden standard" for the performed benchmarks is set as the market leader for LLM inference: NVIDIA's GPU architecture. AWS' *Inferentia* ASIC [8] has been chosen to analyze the differences and compare the performance between the GPU and custom AI chips. The aim of the experiments described in these pages is to answer the following questions:

- Do ASICs — in particular AWS Inferentia 2 chip — actually outperform standard GPUs in terms of the most important performance and business KPIs?
- Which inference hyperparameters have the biggest influence on improving such KPIs on both architectures and which dictate hardware failure or limits?
- Taken into account both considerations, what is the optimal hardware and software configuration to maximize Return Of Investment (ROI) by resulting in the lowest cost per million tokens without compromising the output quality?

Generality in benchmarking is fundamental for generating a result that is meaningful, reliable and can be used to drive business decisions in a wide range of scenarios. In order to achieve a fair and complete comparison, two LLMs of different sizes, manufacturers, architectures and tuning have been chosen. To test vLLM configurations a grid of values has been defined belonging to a set of hyperparameters with limited to no effect on output quality, but rather on hardware load and model performance. Ultimately, every benchmark run consisted of a unique combination of the hyperparameter values, one of

two large language models and one of two hardware accelerators. The result will consist in an empirical analysis based on the collected datasets containing the performance metrics data points recorded throughout the load tests. Finally, a review on costs will determine, according to the assumptions, which configuration and hardware should set the benchmark for LLM inference at any scale.

1.3. Thesis Structure

The following chapters are set to explain in detail and illustrate the various phases of the benchmarking, along with the theoretical notions needed to justify the project's choices. Chapter 2 explores a series of aspects related to how the chosen tools work and how LLM inference is optimized for large-scale deployments from both a software and hardware perspective. Inference is introduced by a brief historical overview followed by an in-depth analysis of its technical details in order to establish a deep understanding of the underlying mechanisms and how changing the settings tunes the execution. Building on this theory, Chapter 3 outlines the project's roadmap and details the specific procedures for cloud instance configuration, model selection, vLLM tuning, load generation and data collection. These sections serve as a comprehensive reference for experiment reproducibility and demonstrate the generality of the methodology. Ultimately, the chapter concludes with a description of the dataset, introducing the empirical foundation required for the subsequent analysis. Chapter 4 describes the processing of the final dataset and presents the outcome of the analyses. It includes an examination of system failures, the interpretation of the resulting visualizations followed by the derived KPI evaluations, and a final analysis translating technical performance into financial efficiency. The *Conclusions and future developments* chapter summarizes the outcomes of the benchmarking, explaining the strengths and limitations of the contributions and proposing future work directions built upon the methodologies and results of the thesis.

2 | Theoretical Background

Statistical Natural Language Processing (NLP) is at the very base of language modeling and manipulation. Early models like n-gram relied on purely statistical, count-based methods. They applied the **Markov assumption** by predicting the next token solely on conditional probability calculated on the preceding $n-1$ tokens. Calculating the probability of a word conditioned on an entire preceding corpus would have been computationally close to impossible. To overcome this limit, the industry shifted to continuous representations in dense vectors and, when Recurrent Neural Networks (RNN) and Long Short-Term Memory (LSTM) were introduced, theoretically, arbitrarily long sequences could be processed and the information could persist throughout the computation. [41] The main flaw of these networks resides in the strictly sequential processing: to compute hidden state h_t, the hardware must first finish computing h_{t-1}. This temporal dependency prevented parallelization across sequence lengths and heavily limited training throughput and context window sizes. The major breakthrough is represented by the introduction of the *Transformer* architecture by Vaswani et al. [45], which dictated a paradigm shift eliminating recurrence entirely in favor of the Self-Attention mechanism, heavily saturating the compute engines thanks to highly parallelized, dense matrix operations. Before these dense matrices can be ingested in the Transformer's attention heads, the raw text must pass through a two-stage input pipeline to be translated into arrays of floating-point numbers.

The input preprocessing begins with tokenization, the procedure by which the human text is split into smaller parts through algorithms like Byte-Pair Encoding (BPE), typically resulting in roughly 1.3 tokens per word in the input sentence. The output of the tokenizer represents a tradeoff between the high granularity of naive character-level splitting and word-level tokenization, which would need an intractable vocabulary size to capture all linguistic variations. Therefore, Subword Tokenization strikes the perfect balance between vocabulary size and vocabulary expressiveness [37].

Discrete token IDs, represented as simple integers, cannot be processed by neural networks. The second step of the input processing is Embedding, the projection of the

discrete token ID into a dense and continuous space. The resulting vector is made of hundreds or thousands of floating point values; each dimension represents a semantic similarity relationship, which in the vector space, is mapped directly to geometric proximity. Finally, the complete input embedding is composed by adding the token embeddings and the positional embeddings. The latter are needed to inject spatial information to the input sequence as the lack of recurrence makes the network permutation invariant [40]. Without positional data, sentences like "there is a rat in the white house" and "there is a white rat in the house" would be treated as equal.

Once raw strings are tokenized, embedded and positionally encoded, the resulting dense input tensors are dispatched to the accelerator's compute cores where a two-phase execution lifecycle, consisting of the Prefill and Decode steps, each bounded by the performance of a different hardware component, generates the final output sequence.

2.1. LLM Generation Architecture

Performing inference for a large language model means predicting one output token at a time given a series of tokens in input. The generation is autoregressive, which implies that each new token depends on the prompt and the history of the previously generated tokens. In fact, the next token is chosen among the ones having the highest probability of immediately following the parts of the sentence already generated, this loop runs until the special <EOS> token is selected and the generation is complete. For each round, from each previous token the model needs to extrapolate the exact semantic information that suits the context; to do so, it uses a mechanism called "self-attention", explained in the foundational paper "Attention is all you need" by Vaswani et al. [45].

Self-attention is a mechanism used to weight how the other words in a sentence contribute to the meaning of a word, or token, in fact, the term refers to identifying the words to which the system literally focuses the attention. It works by assigning to every word in the input sequence three vectors by multiplying the word's embedding by three learned weight matrices:

- **Query (Q)**: what the current word is "looking for" in the rest of the sentence.
- **Key (K)**: the target vector against which the query is evaluated.
- **Value (V)**: the actual semantic meaning of the word.

The attention weights are calculated using the standard scaled dot-product attention

formula:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2.1}$$

where d_k is the dimensionality of the keys (and queries).

Recomputing every Q, K and V tensors each time a new token needs to be generated would represent a heavy computational inefficiency. For this reason, LLMs use a caching mechanism called *KV Cache*, which permits the storage of the Key and Value vectors of the past tokens in the VRAM. It is worth noting that there is no need to cache the Q values as their usefulness is limited to the iteration in which the related token is generated: at each round, the Q, K and V tensors are generated for the current token, the Key and Value are stored in the KV cache and the query is compared to the keys of itself and the other tokens. Finally, the values related to useful keys are fetched to compute the output [26].

When the prompt first arrives, the model enters a first phase named *Prefill*. In this phase the user input tokens are processed concurrently, most of the KV cache content is created and allocated in this phase and — because the processing consists of highly parallel, heavy matrix-matrix multiplications and the memory is accessed for writing ideally once — this operation is compute-bound, which means that the model performance bottleneck is the capacity in Teraflops of the accelerator's compute cores.

The actual output token creation chain happens in the *Decode* phase: processing one new token per sequence, the model performs simpler matrix-vector multiplications, so the bottleneck in this phase is represented by the speed, in GB/s, of the memory bus. The decode phase is therefore considered strictly Memory-Bandwidth bound [33].

To associate these concepts to performance metrics, an inefficient compute-bound prefill primarily affects Time To First Token (TTFT) while a low memory bandwidth determines a high Time Per Output Token (TPOT).

Alongside these issues, the main problem that this thesis addresses is the phenomenon widely referred to as the "Memory Wall", which mainly affects the third fundamental inference performance metric: throughput. In production environments, supporting high user concurrency presents a formidable challenge and the direct cause at inference level is the KV cache management. As highlighted by Kwon et al. [26], KV Cache quickly consumes all available VRAM as it scales linearly with both batch size (the number of concurrent users) and sequence length. This is also where vLLM plays a relevant role by efficiently managing prefill and request preemption, evicting and recalculating inactive requests, mitigating Out-Of-Memory (OOM) failures.

2.2. The Hardware Landscape: GPUs vs. ASICs

With the advent of neural networks, which required the calculation of simple arithmetic operations in several independent iterations, Graphic Processing Units (GPUs) came in handy as they outperformed standard CPUs in matrix multiplications thanks to their highly parallel architectures, with simpler and more numerous cores. The increasing attention to AI computation revolutionized the business of GPU producers, rapidly making machine learning their most profitable market, overtaking the reliance on gaming and traditional graphics [35]. Today, running an LLM on a laptop with a GPU reduces the waiting time for each output token to a fraction of the time needed for the same computation on a CPU. In AI data centers, the same added value is experienced as increased cost efficiency and performance. In AI acceleration, NVIDIA dominates the market as it covered the majority of the share in the last years. The rest of the market is shared between other GPU manufacturers — mainly AMD and Intel — and manufacturers of dedicated chips [9]. These special purpose chips are a type of ASIC: Application-Specific Integrated Circuit, and are created exclusively for AI workloads. They belong to different architectures, such as:

- **TPUs**: Tensor Processing Units, they are popular chips provided in Google's data centers. Their topology is optimized to run AI workloads, both for training and inference. [18]
- **FPGAs**: Field-Programmable Gate Arrays. These components can be reprogrammed to serve specific purposes, making them versatile but specialized, ideal also for Artificial intelligence, they are efficient and cheap, but are not easy to program and need specialized expertise. [24]
- **NPUs**: Neural Processing Units, optimized for neural network inference, they are common in Edge AI for their parallel computing capabilities and limited power consumption. They are increasingly common to be installed in modern chips for personal computers and mobile devices to run local models. [36]

Unlike the ASICs described in the previous list, instead of creating a chip for both AI training and inference, Amazon introduced through the past 10 years two different chips: Trainium and Inferentia. The first one has reached its third generation in 2025 and has been created to address heavy-duty model training, while the second is currently at generation two and is promised to make AI inference cheaper. Given the general availability of the Inferentia 2 chip on different sizes of dedicated AWS EC2 instances — the *inf2* family — and the fact that it is born to optimize inference exclusively, these ASICs, alongside a

comparable GPU on AWS, have been selected to conduct the benchmarking experiments explained in this document. AI ASICs, despite their adoption rates are significantly lower than GPU ones, are facing a rapid increase in the last years. Also more manufacturers are entering the market, breaking the duopoly held by Google and AWS, covering today respectively around two thirds and one third of the AI custom chip share [14]. Their still low adoption can be explained by many problems regarding their design, one of them is the fact that ASICs are created to optimize certain model architectures that represent the standard as of the date they are built, but AI development evolves rapidly, resulting in these chips being obsolete very quickly. The programmability and the market presence of GPUs make the evolution of model architectures faster and facilitate the drift away from ASICs and towards the standard, less powerful, more expensive but more reusable and versatile solution [42].

To complicate the situation even more, NVIDIA has built and spread a deeply integrated software ecosystem to exploit their GPU power at their maximum, through integrations for their proprietary GPU software CUDA and optimizations on the related libraries. Companies rely heavily on this ecosystem, generated by a vendor lock-in that fuels GPU supremacy over the AI silicon industry [16].

ASICs, on the other hand, are often used specifically to counteract this phenomenon. In particular, cloud providers who manufactured an ASIC themselves use them to power their data centers and run AI workloads, both for internal development and features exposed to the public: Google cloud services offer the possibility to make use of virtual machines with TPUs, but also AI services, such as Search, Photos, YouTube and Gemini run mostly on TPUs, which with the last generation to date have also split into two separate architectures: *TPU 8t* for training and *TPU 8i* for inference [13].

AWS is positioned in a similar situation as it exposes EC2 inf2 (and trn3) instances at various scales and also powers internal AI applications such as Alexa and Rufus [15].

Not surprisingly, these companies are the first to benefit a cost reduction after the adoption of their custom solutions replacing the previous GPU-based systems. The aim of the experiments in this thesis is to practically analyze and compare instances from the same cloud provider, having a GPU or an ASIC, reviewing costs but also versatility and simplicity of use.

On Amazon Web Services — the platform chosen to run the tests — the offering for cloud computing instances is wide. For machine learning on GPU, machines belonging to the "P" family are preferred for large-scale training, mounting NVIDIA's top scale chips like the latest "Hopper" and "Blackwell" models (at the time of writing, "Vera Rubin" chips

are yet to be released on AWS), while the "G" instance family offers the most cost-effective deployments for inference and light training, they pack several types and sizes of NVIDIA GPUs from the "Tesla", "A" or "L" GPU families, to account for each need and budget.

2.3. The Role of Inference Engines

PyTorch (the underlying deep learning framework used to execute tensor computations) and Hugging Face (the ecosystem providing the pre-trained model architectures and tokenizer tools) are the essential components needed to serve a Large Language Model (LLM). However, when used directly within a basic Python script, they can only handle one request at a time. To serve multiple users concurrently, you would have to manually implement advanced, ad hoc optimization logic for each specific hardware type. For this reason, inference engines are installed as a middle layer between the model and the machine they run on. These sort of "middleware" help the model to exploit the underlying architecture in the most efficient and powerful way, acting as an operating system for the language model.

Managing Memory allocation, both in standard memory and in the acceleration memory, is the main contribution that engines provide to inference optimization. They embed strategies to efficiently load and distribute the model's parameters and KV cache on the memory of GPUs or ASICs. The next sections will deep dive in the technical aspects of such strategies like PagedAttention and Continuous Batching, that let inference engines juggle multiple user prompts at enterprise production rates.

Indeed, many variants of parallelism are another important benefit of running LLMs on top of these hardware and request orchestration layers. While a simple python script fits for development and preliminary testing purposes, an inference engine is basically necessary to exploit multiple chips, cores, or even machines, and manage thousands of requests by maintaining queues and elaborating many prompts simultaneously.

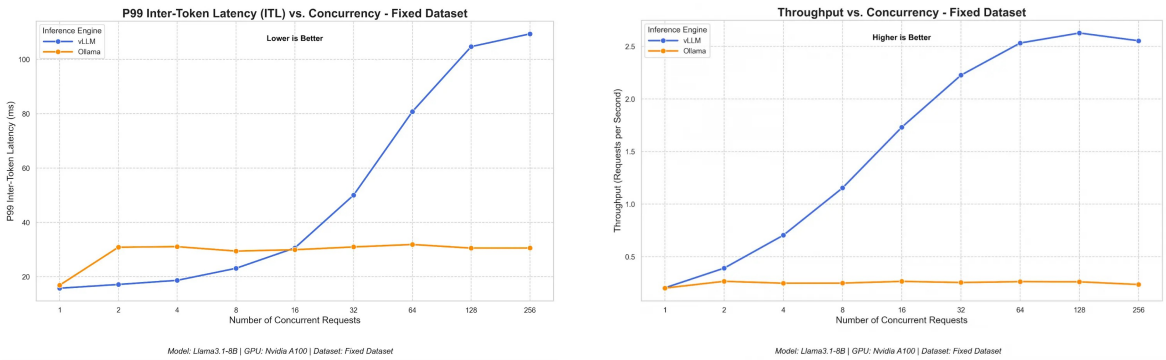
To introduce simplicity of access and transparency, engines expose standardized APIs, often mimicking the OpenAI API, in order to accept and manage stateless requests. Moreover, this black box approach helps making the deployment and upgrade of models seamless to the rest of the code.

The market offers different solutions for different needs also when it comes to inference engines. For personal, small-scale use, the most user-friendly engine is Ollama: a popular application that offers a simple interface and a lightweight backend for serving open source models locally or, recently, in cloud as well. Ollama works with all HuggingFace

models and takes little configuration to serve them and light personalization is possible through the definition of custom system prompts and inference hyperparameters such as the temperature and the input/output lengths. For these reasons, Ollama is the preferred solution for serving small models on CPUs and small GPUs, but struggles with production-scale, high throughput server environments, prioritizing ease of use and often using quantization to fit models on limited hardware.

For high-performance production deployments, an inference engine more robust and capable of heavy-duty work is needed. In these cases the industry-standard solutions are vLLM, developed at UC Berkeley, and TGI (Text Generation Inference) by HuggingFace, with the first being preferred for its optimization for high throughput thanks to the PagedAttention logic [47], while the second is focused on low latency and its native, deep integration with the HuggingFace ecosystem [21].

In these experiments the go-to solution for serving the chosen models has been vLLM, as proven to outperform Ollama in scaling inference to large workloads. Its comparison is featured in many articles, the benchmarks in Figure 2.1 are found in the research "Ollama vs. vLLM: A deep dive into performance benchmarking" published by the *Red Hat Developer* journal [44], and show how vLLM scales while Ollama flatlines both in latency and throughput when increasing concurrency.



(a) Inter-Token Latency vs. Concurrency.

(b) Throughput vs. Concurrency.

Figure 2.1: The images show a benchmark comparison between Ollama and vLLM on different inference metrics.

TGI and vLLM, despite being both valid options for the same types of deployment, present slight differences in performance and, like Ollama, they exceed in specific use cases. For what interests generic enterprise-scale inference, benchmark researches tend to verge their verdict towards vLLM, which excels in terms of throughput and total latency under high concurrency, while TGI specializes in streamlining inference for text generation models,

winning on metric Time To First Token (TTFT) [17]. The research article "vLLM vs Text Generation Inference: LLM Serving Benchmarks That Actually Matter" shows up to date benchmark numbers on these two engines in Table 2.1 and Table 2.2 [1]:

vLLM vs TGI Performance Comparison

Model	vLLM (tok/s)	TGI (tok/s)	vLLM Advantage
LLama-2-7B	15243	4156	3.67x
LLama-2-13B	8934	3187	2.80x
LLama-2-70B (4-GPU TP)	3245	1544	2.10x

Table 2.1: LLM serving throughput benchmarks comparing vLLM and TGI across different model sizes.

vLLM vs TGI Latency Distribution

Metric	vLLM	TGI
TTFT p50	0.24s	0.18s
TTFT p95	0.89s	0.45s
TTFT p99	1.42s	0.71s
Total Latency p50	4.82s	5.91s
Total Latency p95	9.34s	14.28s

Table 2.2: Latency percentile benchmarks comparing vLLM and TGI, highlighting the trade-offs between Time to First Token (TTFT) and end-to-end processing intervals.

These results validate the market dominance of vLLM. Developed at UC Berkeley, transitioned to a VC-backed startup called *Inferact*, raised hundreds of millions of dollars of investments [43], and became the "de facto standard" for enterprise LLM inference [2]. This inference engine is also the fundamental center piece for this thesis' experiments as it permits the configuration of many serving parameters — explained in detail in the following pages — to tune LLM execution at the finest level.

2.4. Inference Software Innovations

Static batching, the technique by which multiple requests are processed in parallel, all beginning and ending at the same exact time, is common in static Natural Language Processing and computer vision, optimal for tasks like image classification or sentiment analysis. LLM serving naively borrowed this technique and was proven to be highly inefficient because, unlike traditional AI in which inputs and outputs have the same length and each batch takes the same processing time, splitting or padding prompts to a standard size is not ideal. Furthermore, an LLM creates one token at a time, this means that all the requests in a batch had to wait for the complete generation of the longest output before finishing computation so that a new batch could begin inference. Given this, static batching for different input and output lengths resulted in massive amounts of idle, wasted GPU/ASIC compute time.

The solution is called Continuous Batching, or Iteration-Level Scheduling, and was introduced in 2022 by Yu et al. [52] within the *Orca* framework, a distributed serving system designed for Transformer-based generative models.

Continuous batching operates at the token iteration level rather than the request level. The GPU/ASIC executes a forward pass to generate one single token for each sequence in the batch, then immediately checks for the presence of End-Of-Sequence (EOS) tokens. For each EOS found, instead of forcing the remaining sequences to wait, it evicts the terminated ones, returns their final output, and deallocates its resident KV cache. Simultaneously, the model pulls new requests from the queue and injects their initial prefill phase in the free slots for the next pass, thus keeping the compute capacity full and active, dramatically increasing sustained GPU/ASIC utilization and boosting the throughput compared to static batching [26, 52]. The vLLM authors Kwon et al. [26] proved the major improvements brought by continuous batching, but also highlighted a collateral effect that led early engines to waste 60% to 80% of VRAM utilization according to the analysis conducted. The phenomenon was caused by the allocation of the KV cache in contiguous chunks based on the maximum possible sequence length, causing severe internal fragmentation: reserved but unused memory for sequences that finish early; and external fragmentation: memory fragmented into small chunks that cannot be utilized.

The proposed solution, which is also the flagship feature of vLLM, is the mechanism of *PagedAttention*. Taking direct inspiration from the classic concept of virtual memory paging in operating systems, PagedAttention divides the KV cache into non-contiguous, fixed-size blocks and maps logical token sequences to these physical blocks through a Block Table. From greedy pre-allocation to avoid running out of KV cache before the

end of the response generation, shifting memory management to PagedAttention led to the complete elimination of the need for large chunks of contiguous memory and, instead, only one small block is allocated for each sequence and the rest would be assigned on-demand the moment the previous fills up, eliminating external fragmentation and reducing internal one. Furthermore, it allows different requests to physically share KV cache blocks, which is mainly useful when multiple requests share contexts or system prompts. By combining Continuous Batching and PagedAttention, modern engines enable systems to safely increase the batch size without hitting the Memory Wall, maximizing throughput and consequently reducing inference costs by enabling precise infrastructure rightsizing, capitalizing directly on the reclaimed memory capacity and maximized computational efficiency.

2.5. Hardware Execution Paradigms

The previous sections explored the aspects of computation that GPUs and ASICs share and reproduce in similar ways, though minor differences apply: during the compute-bound Prefill phase, GPUs dynamically process exact prompt lengths on the fly, whereas ASICs must add a padding to incoming tokens to the nearest fixed boundary, called *bucket*, before executing the dense matrix-matrix multiplications; similarly, during the memory-bandwidth-bound Decode phase, GPUs fetch non-contiguous KV cache blocks via direct thread-level memory pointers, while custom ASICs must orchestrate these sequential matrix-vector retrievals using specialized Direct Memory Access (DMA) engines [8].

Taking into account AWS Inferentia2 as the ASIC architecture analyzed, the main difference in the execution paradigm lies in the compilation of the computational execution graph.

2.5.1. GPU's Dynamic Execution

NVIDIA's GPUs are general-purpose chips, built to handle dynamic, highly unpredictable workloads at runtime as they are highly flexible SIMT (Single Instruction, Multiple Threads) engines. GPU's orchestration relies on the CUDA runtime environment, which executes operations dynamically, meaning that, at each iteration, the CPU decides the physical layout of the GPU's thread hierarchy flexibly, based on the geometry of the matrices representing request input data, instantly launching kernels with entirely different grid, block, and tensor dimensions. This dispatch paradigm enables the precise mapping of physical compute threads to the irregular structure of the inputs managed in continuous batching [31]. Furthermore, this dynamic execution model extends be-

yond compute scheduling to direct memory management. Because general-purpose GPU threads can perform fine-grained pointer arithmetic at runtime, they natively support non-contiguous memory access. During the memory-bandwidth-bound decode phase, custom CUDA kernels allow individual threads to read a request's virtual block table and fetch scattered physical Key-Value (KV) blocks directly into local registers. This direct addressing avoids the need for intermediate memory buffers or dedicated DMA hardware, enabling frameworks like `vLLM` to achieve true zero-padding memory allocation alongside unpadded compute execution [26].

2.5.2. AWS Neuron's Static AOT Compilation

CUDA runtime's general-purpose nature comes at a price: it allows PagedAttention to achieve zero padding, eliminating memory waste, but recalculating the execution structure is computationally demanding, resulting in high energy consumption. AWS Trainium and Inferentia ASICs are specifically built to provide better arithmetic and thermal efficiency, maximizing performance to lower cloud costs. To do so, AWS's proprietary architecture represents a tradeoff between dynamic execution and fixed-size computation paradigms and is called Static Ahead-Of-time (AOT) Compilation. To achieve this efficiency, the hardware relies on fixed, deterministic data paths and specialized hardware blocks like Neuron Cores and custom DMA engines. Therefore, the NeuronX engine must compile the entire computational execution graph Ahead-Of-Time. Receiving at runtime a tensor shape which hasn't been pre-compiled would cause a severe spike in latency as the ASIC needs to pause execution and compile a new execution graph. The workaround is represented by sequence length "Bucketing": to prevent runtime recompilation while serving variable user prompts, the runtime forces incoming sequences into predefined, pre-compiled dimension boundaries, managing reduced fixed sequence lengths (e.g. of 64, 128, or 512 tokens). AOT graph compilation reintroduces internal fragmentation to a minimal extent, as user prompts shorter than the bucket dimension need padding with unused tokens. [5] Notably, while Inferentia2 accelerators introduce native hardware-level support for dynamic shapes to improve upon *Inferentia1*'s strictly static constraints, production serving frameworks continue to optimize execution by pre-compiling static graphs across power-of-two sized buckets, dynamically switching between these discrete boundaries to minimize the High-Bandwidth Memory (HBM) memory bus overhead when retrieving the resident KV cache. [6]

To synthesize the architectural and operational trade-offs explored throughout this section, Table 2.3 provides a comparison between general-purpose NVIDIA GPUs and custom AWS Inferentia2 accelerators.

Accelerator Execution Paradigms Comparison

Architectural Dimension	NVIDIA GPU	AWS Inferentia2
Execution Paradigm	SIMT with dynamic, JIT dispatch	AOT static graph compilation
Input Processing	Sequence dimensions processed on the fly; zero padding	Inputs padded to precompiled buckets
Memory Access	Direct thread-level pointers via custom CUDA kernels	Dedicated DMA hardware engines
Fragmentation	Near-zero internal waste via PagedAttention	Controlled internal waste introduced via padding
Runtime Overhead	High host-side CPU dispatch and scheduling energy	Minimal, avoids recompilation through bucketing

Table 2.3: Summary of architectural and execution divergences between GPUs and AWS Inferentia2 during LLM serving.

2.6. Open-Source LLM Architectures

Large Language Models like Google Gemini, OpenAI ChatGPT, or Anthropic's Claude, represent the highest level of innovation for the respective manufacturer, but as closed-source, they are only accessible through the provided graphical user interfaces and abstracted web APIs. Their personalization through additional fine-tuning and hyperparameter tuning is limited, as well as their underlying architecture. Given this, they also intrinsically need to be accessed online and there is no possibility for locally hosting them. For these reasons, and for a better cost management, companies often opt for open-source models, where exact weights and architectures are publicly available, and consequently they are the only ones available for serving on vLLM [3]. These solutions provide absolute programmatic control for companies to expose their personalized models and have total authority over memory management, computational capacity, data regulation and cost containment. The open-source LLM offering evolves daily, with new models being released from independent users, small AI companies and tech giants globally. The most popular open-weight LLM families include the *Gemma* models by Google DeepMind, Meta's *Llama*, Alibaba's *Qwen*, and models from *DeepSeek* and *Mistral AI*. Millions of models

are published and accessible on public repositories such as *Hugging Face*, the largest hub for machine learning, or GitHub, OpenML, and specialized repositories for their respective ecosystems like TensorFlow Hub and PyTorch Hub. For the experiments in this thesis, the two models chosen are *meta-llama/Llama-3.2-1B* [3] and *Qwen/Qwen2.5-3B-Instruct* [51] available on Hugging Face, vLLM's default model repository [50]. These two models have been chosen to maximize the generality in the benchmarking by avoiding specific aspects that might be tied to the underlying architectures rather than the execution environment. Notably, the choice has been severely limited due to hardware compatibility: while CUDA runtime on NVIDIA GPUs supports all HuggingFace open-source models, AWS Inferentia2 natively supports a limited set of mathematical operators used in the models' layers, such as Grouped-Query Architecture, specific activation functions and Rotary Position Embeddings, thus limiting the choice to the model families listed in the Hugging Face documentation for AWS Trainium and Inferentia [20]. The following subsections explore the theoretical details of open-source models and justify for each one the contribution to result generality provided by the selection of the two models, which in the following pages for simplicity will be referred to as **Llama**(meta-llama/Llama-3.2-1B) and **Qwen**(Qwen/Qwen2.5-3B-Instruct). The details included can be found in the Qwen2 Technical Report by Yang et al. [51] and the corporate report The Llama 3 Herd of Models [3].

2.6.1. Serving Infrastructure Control

Memory-level optimizations, such as PagedAttention, require full control over model architectures, bypassing API abstractions entirely to directly manipulate physical memory buffers.

- **Llama**: The Llama 3.2 family is focused heavily on edge AI and is highly optimized for efficiency in limited hardware. It is licensed under the Llama 3.2 Community License, which allows for broad commercial use but has specific, custom limitations. It uses GQA (Grouped-Query Attention) to reduce memory overhead, unlike the previous versions, which used the conventional multi-head attention (MHA).
- **Qwen**: Licensed with the standard Apache 2.0 license, its family focuses on specialized expert models for text generation, coding and math reasoning. It also offers superior multilingual support. Uses GQA as well, transitioned from MHA starting with the Qwen2 generation.

Both model architectures include *SwiGLU*(Swish-Gated Linear Unit) activation functions rather than standard ReLU, which provide better gradient flow and non-linearity for deep

models, and RMSNorm (Root Mean Square Layer Normalization) pre-normalization before each transformer block to improve training stability. Testing across two distinct corporate architecture paradigms proves the benchmark to be framework-agnostic, making memory optimization rely successfully on standard structural access rather than overfitting on the model formats.

2.6.2. Training Paradigms

Base models, also known as foundation models, undergo a causal pre-training with massive datasets of trillions of tokens from web crawls, books, encyclopedias and other sources. These models generate as response the sequence which purely resembles the most likely output given the input prompt, following the patterns learnt from the training set, usually trying to complete the input sentence, as their training consists of predicting the next token in the sequence. They represent a generic base for many Natural Language Processing tasks and for further fine-tuning. On foundational models, usually to turn them into actual chatbots, an additional supervised fine-tuning step is executed in order to moderate the model's responses to match a conversational register and ensure a user-friendly interaction.

- **Llama**: The chosen model is a general-purpose, foundation model. Instruction-tuned versions of Llama 3.2 exist.
- **Qwen**: As the added suffix *"-Instruct"* in the name suggests, this is the instruction-tuned version of the related foundation model. It relies on explicit chat-template structures provided in the `tokenizer_config.json` file.

In order to use a foundation model as a chatbot, an additional configuration — a `.jinja` file called *chat template* — is provided, enabling the model to identify the system prompt, the query and the instructions regarding the roles in the conversation. This additional configuration enables the model to be queried by the load generator for correctly executing the benchmarks.

2.6.3. Model Dimensions

The model parameter count includes the number of neuron's weights and biases in the neural network components across several different architecture layers. A higher count generally indicates better complex pattern comprehension but requires higher computational power and linearly proportional memory capacity. The VRAM occupation, not considering the dynamic KV cache, is determined also by the type of the parameter val-

ues (dtype), which are usually represented as 16-bit or 32-bit floating-point numbers, with the 32b dtype effectively doubling the required memory with respect to the 16b format.

- **Llama**: Includes around 1.23 billion total parameters, 16 layers with hidden layer size 2048, Intermediate size 8192 (MLP), 32 attention heads (GQA) and 8 KV heads.
- **Qwen**: Includes around 3.09 billion parameters, 36 layers with hidden layer size 2048, Intermediate size 11008, 16 attention heads and 2 KV heads.

Serving different sized models enable to empirically understand to a minimal extent how vLLM memory management and computation scale with the number of parameters and how the various hyperparameters contribute to performance capping for different model sizes in fixed-size hardware.

2.6.4. Prompt and Context Processing

The first step in LLM inference is the input preprocessing: the full prompt, bounded by the dimension of the context window, is divided in tokens by the tokenizer, each token represents a symbol, punctuation or usually part of a word, depending on the model's vocabulary. The tokens are then translated into token embeddings and positional embedding is added to compose the final input (input embedding).

- **Llama**: The 3.2 version inherits the Llama 3.1 tokenizer. Its vocabulary is made of approximately 128,256 tokens, heavily weighted towards English and Western Europe Languages.
- **Qwen**: Its tokenizer has a larger vocabulary of 151,643 tokens, optimized for high multilingual efficiency, particularly for Asian languages, allowing it to represent the same information with fewer tokens.

Both models support 128k token long context windows. They also share the same Rotary Positional Embeddings (RoPE) representation.

Evaluating multiple model architectures ensures that the benchmark results are generalized rather than tied to a single architecture. Introducing diverse model geometries decouples the hardware performance profiles from model-specific biases, tokenization schemes, or training artifacts. This methodology helps to attribute generality to the fact that the observed scaling patterns in throughput and latency reflect fundamental transformer-serving dynamics under system stress, rather than localized efficiencies unique to a single model run.

3 | Experimental Methodology

This chapter explores the methodologies used to set up and run the LLM benchmarking on cloud instances. The following sections introduce the tools, environments, machines, configurations, models, metrics, programs and scripts that enabled the execution of rigorous and reproducible experiments. The project choices, the executed operations and the best practices applied will be described and justified, providing a complete understanding of each phase of the dataset collection. The project's experience also encompasses the mistakes that were made, which are cited throughout the paragraphs to provide the reader with the contextual knowledge acquired during troubleshooting.

As stated in the introduction, the experiment's objective is to execute an empirical validation of cost and performance on two primary axes of optimization for self-hosted Large Language Model serving:

- The physical runtime environment, represented by the experimentation on two different hardware architectures: NVIDIA's general-purpose GPU, the market's gold standard for machine learning workloads; versus the Inferentia2 chip, the ASIC developed by AWS specifically for AI inference.
- The logical inference engine configuration, consisting of a series of vLLM hyperparameters by which the system performance can be tuned.

The comparison is intentionally treated as an infrastructure-level test, evaluating the combined synergy of the silicon chip, the host compiler and the serving framework acting as a unified entity.

Through deterministic workload generation, the performance testing aims to simulate real-world, highly parallel user request loads to produce reliable results, capable of modeling real enterprise use cases and determining the contribution of each design choice to achieving the desired AI inference Service Level Objectives.

This chapter proceeds by detailing the environments on which the tests have been executed by including each cloud instance technical specification, along with its setup and management recommendations.

Secondly, a comprehensive examination of the model serving configuration with vLLM is provided, followed by the load generator's modalities and a focused description of the purpose of each hyperparameter in the search grid. A section dedicated to the additional software used follows. This software has been embedded in the procedures during the test setup to guarantee a robust and fault-tolerant benchmark execution.

The test code and the supporting script references are built gradually as the components are presented and their purposes explained, culminating in the detailed dataset description, which completes the necessary knowledge to comprehend the subsequent result review chapter.

3.1. Infrastructure and Test Environments

The two specific machines chosen for the tests are AWS inf2.xlarge and g4dn.xlarge EC2 instances. They represent the entry level virtual machines for model inference and they have similar technical data sheets, with the main difference being the silicon at the base of their infrastructure: the Inferentia2 ASIC, powered by Amazon's *NeuronX* cores, mounted by the inf2.xlarge; and the NVIDIA T4 GPU, on the g4dn.xlarge.

3.1.1. Cloud Instance Types

The "G4dn" family has been selected to conduct the benchmarks, featuring entry-level NVIDIA T4 GPUs. The family offers a variety of VM sizes to account for the different cost and power needs the user might have. Table 3.1 illustrates the details of each G4dn instance sizing, from the single-GPU, PC-sized machines to the enterprise-scale, multi-GPU ones.

AWS G4dn Instance Specifications

Instance Size	GPU	vCPUs	Memory (GiB)	Storage (GB)
Single GPU VMs				
g4dn.xlarge	1	4	16	1 x 125 NVMe SSD
g4dn.2xlarge	1	8	32	1 x 225 NVMe SSD
g4dn.4xlarge	1	16	64	1 x 225 NVMe SSD
g4dn.8xlarge	1	32	128	1 x 900 NVMe SSD
g4dn.16xlarge	1	64	256	1 x 900 NVMe SSD
Multi GPU VMs				
g4dn.12xlarge	4	48	192	1 x 900 NVMe SSD
g4dn.metal	8	96	384	2 x 900 NVMe SSD

Table 3.1: Specifications for EC2 G4dn instances featuring NVIDIA T4 GPUs. Source: `https://aws.amazon.com/it/ec2/instance-types/g4`

In the tests with GPU, the instance type which has been used is g4dn.xlarge. Its cost is €0.559 per hour for the *On demand* tier, and €0.20/h for the *Spot* tier [22]. The first one being the tier for the always available machine, the second is a special AWS instance type subject to limited, not guaranteed availability and because of that its price is lower. The On-demand tier has been selected for the benchmarks in order to avoid instance interruptions and ensure the full execution of all tests. Prices are considered for region "eu-central-1" and with Linux operating system.

As stated previously, the choice for the machine used for tests on ASIC has been the proprietary AWS "Inferentia 2" family, in particular a "inf2.xlarge" instance, which costs €0.967/h for the On demand tier, and €0.152/h for the spot tier [23]. For reference on scale, the inf2 instances specifications are detailed in Table 3.2.

AWS Inf2 Instance Specifications

Instance Size	Chips	Acc. Mem	vCPU	Mem (GiB)	Storage
inf2.xlarge	1	32 GB	4	16	EBS Only
inf2.8xlarge	1	32 GB	32	128	EBS Only
inf2.24xlarge	6	192 GB	96	384	EBS Only
inf2.48xlarge	12	384 GB	192	768	EBS Only

Table 3.2: Specifications for EC2 Inf2 instances powered by AWS Inferentia2 chips. Source: `https://aws.amazon.com/it/ec2/instance-types/inf2`

In general, scaling up the AWS Elastic Compute Cloud instances while keeping the same instance family translates into selecting among instance sizes that include an increasing number of identical accelerators. Both vCPU counts and memory capacities multiply their original value by the same number, reported in the instance type name before "xlarge". Therefore, the inf2 family spans from inf2.xlarge to inf2.48xlarge, with the latter having 192 vCPUs, 768 GiB of memory and 12 Inferentia2 chips. In contrast, g4dn offers sizes up to g4dn.16xlarge — 64 vCPUs and 256 GiB — and g4dn.metal — 96 vCPUs and 384 GiB — with only the latter and g4dn.12xlarge having more than one 16 GB NVIDIA T4 chip, counting respectively 8 and 4 GPUs each.

Chosen to be comparable, the two selected instance types integrate the same number of virtual CPUs — 4 — and the same memory size — 16 GiB — but differ in some architectural aspects. For example, the inf2.xlarge runs on an AMD EPYC physical processor and has a clock speed of 2.95 GHz, while the g4dn.xlarge reaches the frequency of 2.5 GHz and is based on the Intel Xeon CPU family. Furthermore, the accelerator's memory — also called *Video Memory*, a legacy nomenclature reflecting the historical origins of GPUs as dedicated rendering pipelines — has a capacity of 16 GiB on the g4dn.xlarge and it doubles on the inf2.xlarge [22, 23]. These differences don't invalidate the experiments, but they will be considered in the analyses and are partially justified by the difference in price, which is €0.559/h, for the GPU instance and €0.967/h for the ASIC one. Both prices are meant for the On-Demand deployment type and the Linux Operating System. The g4dn offer includes the option for Windows, subject to a price increase.

Table 3.3 summarizes the hardware specifications for the two machines on which the experiments were run.

Evaluated Hardware Environment Summary

	g4dn.xlarge	inf2.xlarge
Accelerator Silicon	NVIDIA T4 GPU	AWS Inferentia2 ASIC
Accelerator Cores	NVIDIA Tensor Cores	Amazon NeuronX Cores
Video Memory	16 GiB GDDR6	32 GiB HBM2e
vCPUs	4	4
System Memory	16 GiB	16 GiB
Physical Processor	Intel Xeon Family	AMD EPYC 7R13
Clock Speed	2.50 GHz	2.95 GHz
Network Performance	Up to 25 Gbps	Up to 15 Gbps
On-Demand Price (Linux)	€0.559/h	€0.967/h

Table 3.3: Hardware specifications summary for the evaluated AWS EC2 instances.

For the model compilation on Neuron, a machine with the same chip but with more compute power and memory was needed, as detailed in the next section. The instantiated host was of type *inf2.8xlarge*, which has 32 vCPUs and 128 GiB of memory.

3.1.2. The Machine Image

During the cloud instance setup, AWS provides the user with a selection of plain operating systems as well as a series of Amazon Machine Images (AMIs): preconfigured virtual machine templates used to launch instances with ready-to-use drivers, libraries and software for different specific use cases. For what concerns Machine Learning, the Deep Learning AMI (DLAMI) [11] is suggested, as it natively includes validated, production-grade versions of the NVIDIA Driver, the CUDA Toolkit and low-level communication primitives like cuDNN and NCCL. The DLAMI, nowadays, is typically based on Ubuntu 22.04 LTS. Moreover, AWS pre-tunes the host kernel parameters inside these AMIs specifically for heavy Input/Output and compute workloads, such as LLM inference.

For the Inferentia and Trainium machines, AWS provides a customized operating system baseline known as the *Deep Learning AMI for Neuron* [7], replacing the CUDA ecosystem with the AWS Neuron SDK. Launching an inf2.xlarge with the DLAMI pre-installs the low-level kernel driver `aws-neuronx-dkms` that enables the mapping of physical page buffers and DMA requests to the NeuronX silicon cores, alongside the dedicated runtime library `aws-neuronx-runtime-base` and the compilation toolchains `neuronx-cc`. Using

this standard base image eliminates host-side configuration drift between the benchmark iterations and ensures the runtime environment is optimized for the exact underlying hardware.

3.2. vLLM Configuration

Alongside the chosen infrastructure, the inference engine is the focus point of the benchmark. It elevates model serving, compared to a raw python execution, by introducing a layer that acts like an operating system for the LLM. It exposes OpenAI-like APIs, manages the models' download and storage, and orchestrates memory and silicon to efficiently serve multiple parallel requests.

vLLM represents the industry standard for enterprise LLM inference. In its standard configuration, it is as practical to install and use as any common package on Linux, but it offers a high level of customizability as well.

3.2.1. Installation

The vLLM version used for the tests is `v0.18.0`. The simplest method to install is through the *uv* or *pip* package manager. Other possible installation procedures include building from source through GitHub, the slowest but most customizable installation to help dealing with environment's CUDA and Python version mismatches, or installing using the official Docker Images, allowing for a containerized setup with pre-installed necessary dependencies. For installations on specialized hardware such as AMD GPUs, Apple Silicon, and accelerators like Google Cloud TPU and AWS Neuron, vLLM also provides several *wheels* and installation instructions.

Resolving dependencies has been the main issue faced in the setup phase. Eventually, all standard installation methods have been tried with little success, the cause turned out to be the version incompatibility between the fresh vLLM and the dated software pre-installed on the g4dn.xlarge instance.

vLLM operativity achievement has been significantly easier since the DLAMI for Neuron [7] includes a python virtual environment with the service engine already present. This means that, between the inf2.xlarge startup and a successful `vllm serve` command, the only other bash command to be run is

```
source /opt/aws_neuronx_venv_pytorch_inference_vllm/bin/activate
```

3.2.2. Model Serving

There are several ways to import a model for vLLM serving. The simplest one consists of providing the model name as found in the Hugging Face Hub (e.g., `meta-llama/Llama-3.2-1B`). After running the `vllm serve` command, the engine autonomously retrieves and downloads the model from the repository, and stores in its default directory the weights and configurations by depositing the `model.safetensors` and `config.json` files, containing respectively the model's parameters and the inference engine instructions, comprising serving hyperparameters and other vLLM configurations.

The use of some models on Hugging Face can be governed by a custom, commercial license agreement, which implies that the model can be downloaded after the acceptance of the license agreement in the Hugging Face website. The user authentication is configured by the generating a personal token to be included in the `HF_TOKEN` environment variable.

Alternatively, if the model is a custom fine-tuned version or has been previously loaded from a different source, the path containing the model.safetensors and config.json file can be provided instead of the Hugging Face model name.

The optional model hyperparameters can be declared as parameters of the `vllm serve` command (e.g., `--max-num-seqs "128"`).

The complete vLLM model serving command figures as follows:

```
vllm serve "\$model" \
--port 8000 "\${engine_args[@]}" \
--enforce-eager \
--chat-template /home/ubuntu/chat_template.jinja
```

The `--enforce-eager` parameter forces the engine to run in eager mode, which disables the allocation of CUDA Graphs, a system that trades more memory consumption for less execution overhead by pre-allocating GPU for the entire execution path. Disabling this system enables the saving of typically 1 to 2 GigaBytes of VRAM, which on 16GB of VRAM can help reduce the risk for OOM errors. Moreover, this parameter is recommended when serving vLLM on Inferentia2 as it doesn't rely on CUDA graphs. Avoiding the insertion of `--enforce-eager` when running the command would result in errors or cause the vLLM server to hang.

When the `vllm serve` command is first run, a sequence of memory-intensive stages takes place: vLLM loads the weights from the provided path or loads the model from Hugging Face, runs a profiling pass to scan the remaining memory, calculates how many KV cache

blocks can fit and consequently allocates almost all available VRAM into a KV Cache Pool, according to the `--gpu-memory-utilization` parameter and the memory management paradigm of the underlying architecture. During the model serving, mostly in the startup phase, the system can fail due to an Out Of Memory (OOM) error. The failure can occur initially when the model which the system is trying to load has weights that take more space than the available memory capacity, such as trying to load an 8 billion parameter model, which by itself occupies around 16GB of VRAM in `float16` weights and biases, saturating the entire g4dn.xlarge's T4 memory. A `RuntimeError` can be produced due to insufficient memory if the profiling scan does not detect enough space for allocating a single KV cache block. Lastly, the memory limit can be exceeded by the allocation of the total calculated KV cache size concurrently with memory utilization from an external process, or by a memory leak caused by external fragmentation [26].

3.2.3. Model compilation on Neuron

In addition to the previously listed cases, further OOM episodes were faced during the startup phase, caused by the RAM-intensive model compilation performed by vLLM on the inf2.xlarge instance, which have not been resolved even reducing the chosen model size to 1B from an initial 4B, then 2B model. In fact, in order to let vLLM take advantage of specialized ASIC hardware for inference, the model needs to be compiled Ahead-Of-Time. The compilation consists of the use of Accelerated Linear Algebra (XLA) to transform the model into a high-level compute graph before lowering it to hardware-specific machine code: the NEFF file [4]. The intermediate tensor, during the compilation, can occupy a large amount of Host RAM, which can cause OOM errors, even with the 1B parameter model meta-llama/Llama-3.2-1B on 16 GB of standard memory.

The solution to the problem is straightforward: compile the models on a machine with more memory and, given the strict hardware compatibility of the pre-compiled model files, an instance type of the inf2 family was needed. The next tier in its catalog is inf2.8xlarge, which includes a 128GB host memory, as well as 32 vCPUs and the exact same ASIC as the inf2.xlarge.

Configuration-wise, the model graphs calculated vary with the values of some architecture-related hyperparameters which, for the purpose of the benchmarks based on the hyperparameter search grid described later in this chapter, meant that a model version had to be compiled for each combination of the values of hyperparameters dtype, max_num_seqs and max_num_batched_tokens. Each parameter in the reduced grid for model compilation has 3 values, resulting in 27 `vllm serve` runs for each of the 2 models, which

have been executed by a dedicated python script (`neuron_compile_models.py`, found in the Appendix). The models, saved in a directory on the compilation machine, have been loaded on the smaller inf2.xlarge machine in a directory called `all_compiled_artifacts`. Each of its subfolder, named according to the convention

```
dtype_{dtype}_seqs_{seqs}_batched_{batched_tokens}
```

included a `model.pt` and a `neuron_config.json` file. The first file contains the NEFF information, representing the serving modalities for the related model and specific vLLM configuration; the latter is the collection of the serving hyperparameters and the instructions for vLLM on how to load the model. The compilation cost needs to be considered when first serving a model in these instances since the model's one-time compilation on a more expensive machine can increase the cost for the initial model deployment: the inf2.8xlarge has an on-demand hourly price of €2.51, more than two and a half times the inf2.xlarge price. This charge will not be part of the financial analysis as it does not represent a running cost.

From the inf2.8xlarge machine, also the model.safetensors and config.json have been transported to the inf2.xlarge. Alternatively, these two files could be generated by triggering the download from Hugging Face of the same model on the destination machine through a subsequently failing `vllm serve`.

To switch between the compiled model configurations when executing the benchmarking on the Inferentia2 machine, a launch folder has been created and, separately for each model, such folder contained the following items:

- `model.safetensors`: The trained parameters of the considered model.
- `tokenizer.json`: Containing the vocabulary, the tokenizer translates the prompt into tokens and provides them to the model.
- `model.pt` symlink: A symbolic link (Linux symlink) that points to the model's specific graph description.
- `neuron_config.json` symlink: a symbolic link pointing to the model configuration description.

Thanks to the use of symlinks, when cycling through the search grid, the script reassigns the paths to the currently tested configuration, enabling vLLM to detect the correct files and to avoid model re-compilation before a benchmark run.

3.3. The Load Generator

Executing a benchmark requires the synthetic generation of a fixed, reproducible workload submitted to the target system to evaluate its performance characteristics relative to an established baseline.

GuideLLM [30], originally developed by Neural Magic and later integrated into the official `vllm-project` ecosystem following its acquisition by Red Hat, is an automated, open-source load-testing framework explicitly designed for Large Language Models. Unlike generic HTTP benchmarkers that strictly measure raw network request and response latencies, GuideLLM is architecture-aware. It tracks LLM-specific telemetry such as Time to First Token (TTFT) and Time Per Output Token (TPOT), providing deep visibility into the distinct prefill and decode phases.

The generated workload simulates a sequence of distinct inference queries transmitted by concurrent users to the OpenAI-compatible endpoint exposed by vLLM.

The generation profile can be adjusted via the configuration arguments of the `guidellm benchmark` command as detailed below:

- `--target`: the HTTP endpoint on which the vLLM service to be tested is exposed. Because the inference engine and the load generator were hosted locally on the same instance during these experiments, this parameter was configured to point at `localhost`.
- `--profile`: previously named *rate-type*, defines the pattern by which the traffic is generated, the options include the pure sequential (`synchronous`) or parallel (`concurrent`) execution, a specified fixed flow of requests per second (`constant`), the maximum capacity setting (`throughput`), the sending at a randomized rate (`poisson`), or the automatic rate exploration (`sweep`).
- `--rate`: this parameter's meaning depends on the previous and tunes its baselines or limits. For `--profile sweep` it represents the number of benchmarks executed with various `req/s` rates, in the concurrent execution profile it defines the number of simultaneous requests, i.e., the number of simulated concurrent users, while for the constant and poisson profiles the rate provided is the actual value in requests per second. For the remaining options of `--profile`, the `--rate` parameter doesn't affect the execution since, for the `throughput` setting, the maximum capacity is calculated automatically while in the `synchronous` profile a new request is sent at the fulfillment of the previous.

- `--max-seconds or max-requests`: limits the single benchmark's execution duration by defining a maximum number or requests sent or the maximum run duration in seconds.
- `--data`: this parameter, along with the optional `--data-args`, `--data-column-mapper`, `--data-samples` and `--processor`, defines the request dataset source and configurations. It accepts HuggingFace dataset IDs, local file paths or synthetic data configurations in *key=value* pairs.

For the experiments in this thesis, the `sweep` load generation profile has been chosen to automatically identify the stable operating ranges and performance ceilings. Firstly, it executes a synchronous run, followed by a maximum capacity one, the remaining are constant-rate runs spanning the safe range identified. A limit of 60 seconds for each run has been set.

The dataset configuration has been set to fixed-length synthetic requests and output sentences to directly control the duration of the decode phase and measure the hardware's raw sequential processing speed over a sustained period. The `prompt_tokens` and `output_tokens` values have been defined empirically to mirror real, simple request lengths and to ensure the completion of the benchmark runs given the duration threshold.

The GuideLLM benchmark launch command used in the bash script results as following:

```
  guidellm benchmark \
--target http://localhost:8000 \
--profile sweep \
--rate 10 \
--max-seconds 60 \
--data "prompt_tokens=400,output_tokens=400"
```

During runtime execution, GuideLLM outputs a live terminal matrix reporting the status and live metric data for each benchmark run, as well as the list of subsequent runs and the expected remaining execution time.

Once the benchmark terminates, the console prints a statistical summary aggregation.

The definitive, authoritative output of the test is exported into a structured `benchmarks.json` data container storing comprehensive system metadata, per-run latency percentiles, and request timestamp breakdowns. A corresponding `guidellm.html` report is output for interactive browser visual analysis.

Following the completion of an evaluation cycle across the active vLLM hyperparameter search grid, the telemetry stored within `benchmarks.json` was extracted and merged with the metrics captured by the custom monitoring setup detailed in Section 3.5, forming the data foundation for the analyses in this work.

3.4. Experimental Pipeline

3.4.1. The Iterative Process

The bash script was developed to run the multiple benchmarks and data collection pipelines automatically. It has been revised through the project to minimize human intervention in case of unexpected failures, such as an OOM error, a vLLM startup crash or GuideLLM stalling. Whenever a re-execution was needed, a check for the already completed benchmarks, by verifying the presence of the specific output files, enabled the program to skip such runs, resume the search and repeat the benchmarks that did not conclude to ensure the failure for the specific configuration is deterministic and architecture-related.

After activating the Python virtual environment and setting the environment variables such as `HF_TOKEN` for HuggingFace authentication, the code defines a series of resiliency helper functions to enhance logging, execute retries after failures, and kill the background processes between the iterations or as final cleanup. The iterative functioning consists of cycling through a predefined hyperparameter search grid and, for each combination, serve vLLM on the unique set of parameter values, initiate the data collection system, run the GuideLLM benchmark and, once finished, process and aggregate the collected telemetry from the different sources into a single dataset. As the model graphs were saved on the internal file system into ad-hoc labeled directories, the version of the script executed on the inf2.xlarge instance included the logic to retrieve the model pre-compiled on the current hyperparameters and reassign the symlinks in the launch directory to the correct `neuron_config.json` and `model.pt` files.

3.4.2. The Hyperparameter Search Grid

To explore the performance boundaries of the hardware backends and map an efficient frontier for throughput and latency, the following vLLM inference engine arguments [48] were altered:

- `--max_num_seqs`: dictates the number of concurrent active requests that the iteration-

level scheduler can batch together in a single forward pass. Increasing this parameter optimizes the GPU/ASIC core utilization but causes higher queuing latency and memory stress by regulating the scheduler's ability to interleave the prefill phase of new streams with the decode phase of existing ones. The `--max_num_seqs` parameter values considered for the experiments were 64, 128 and 256.

- `--max_num_batched_tokens`: the maximum number of batched tokens per iteration. It sets the ceiling for the number of tokens processed across all active sequences in a single prefill, preventing the stalling of ongoing decode streams. In the grid, this parameter has been set to assume values 4096, 8192 and 16384.
- `--dtype`: defines the data type of the model weights in terms of floating point precision. By modifying this value, the memory footprint and compute intensity is altered. vLLM accepts 3 possible values for `dtype`:
  - `float32` (or `float`): the standard floating point precision, structured as 1 sign bit, 8 exponent bits, 23 fraction bits, requires 4 bytes for each model parameter.
  - `float16` (or `half`): 1 sign bit, 5 exponent bits, 10 fraction bits. Less precise than float32, but occupies half the memory, allowing for more KV cache allocation.
  - `bfloat16`: 1 sign bit, 8 exponent bits, 7 fraction bits. maintains the float32 dynamic range by dropping the number of precision bits. This format has been developed by Google Brain to solve the stability flaws of float16 during deep learning workloads [25]. On the g4dn.xlarge instance, this parameter has not been tested as it is only supported on GPUs with a higher compute capability score of 8.0 against the 7.5 of the T4 GPU.
- `--gpu_memory_utilization`: defaulting at 90%, this parameter determines the fraction of total GPU memory that vLLM is authorized to reserve for model weights and the KV cache pool, leaving a memory buffer for runtime allocations and compilation overheads. By fitting more KV cache onto the accelerator's memory, the inference performance is enhanced, but the system is more prone to OOM failure. In Neuron architectures, due to static AOT graph allocation, gpu_memory_utilization is ignored as the memory blocks are predefined during compilation via the `neuron_config.json` boundaries. On the GPU machine, the values considered for this hyperparameter are 0.80, 0.85, 0.90 and 0.95, fractions of these values are set in environments in which the GPU is shared among multiple models or servers.

Between the experiments, one of the hyperparameters in the original search grid —

`--swap_space` — has been deprecated, reducing the number of benchmarks considered in the analyses as no effect was produced by varying its value. This parameter, ignored in the previous version and later returning an initialization error if set, was removed during the V1 engine transition, with the release of vLLM version 0.18.0, as it was primarily associated with the legacy `best_of` sampling parameter. Swapping on the host RAM is now replaced by weight offloading (allocating a fixed amount of host RAM for model weights exceeding the VRAM size) and request preemption.

The final hyperparameter search grid is represented in Table 3.4.

Hyperparameter Search Grid

Hyperparameter	Explored Space
max_num_seqs	64, 128, 256
max_num_batched_tokens	4096, 8192, 16384
dtype	`float32`, `float16`, `bfloat16`
gpu_memory_utilization	0.80, 0.85, 0.90, 0.95

Table 3.4: Experimental hyperparameter search grid configuration showing execution scopes across both dynamic GPU and static ASIC hardware environments.

The combination matrix produced 27 combinations for the inf2.xlarge and 72 for the g4dn.xlarge. Multiplying these grid combinations by the two evaluated LLMs yields a total of 198 independent benchmark runs, which, lasting around 20 minutes each, were executed in a combined effective time of 49 hours and 30 minutes. The actual elapsed time for the experiments differs as it takes into account the variations caused by multiple retries for the failed runs and the 54 model compilations on inf2.8xlarge. Furthermore, to optimize the time taken for the experiments, as the EC2 instance price wouldn't vary being billed hourly, one inf2.xlarge and two g4dn.xlarge instances were utilized in parallel, with the latter two running benchmarks each on half of the search grid.

3.5. Telemetry Infrastructure

Metric collection from several heterogeneous observability vectors is crucial to establish comprehensive visibility of the system performance both at infrastructure and application level. The monitoring system was designed as a background process initiated concurrently with the vLLM server deployment, ensuring a continuous time-series record through the

load generation of each benchmark. The telemetry layers recorded a sample of each metric every 5 seconds; this sampling interval produced an average of 130 data points per experiment, sufficient to analyze in depth the system's reaction to the synthetic load sweep. The main monitoring source is the Prometheus service provided by vLLM [49] at endpoint `http://localhost:8000/metrics`, from which the inference engine-related metrics are polled in the Prometheus *Counter*, *Gauge* or *Histogram* formats. These metrics include the main latency and throughput measurements, as well as the KV cache handling statistics and prefill/decode times and token counters. The infrastructure metrics, such as the accelerator's memory and compute utilizations, are extracted with the use of the hardware-specific telemetry tools natively integrated in the two instances. For NVIDIA GPUs, the authoritative data source identified is `nvidia-smi` (NVIDIA System Management Interface) [32], which acts as a low-overhead telemetry daemon to track and query the operational state of the GPU. Conversely, on the inf2.xlarge, the monitoring module deployed on the AWS Neuron instances is an asynchronous command-line tool that streams continuous telemetry natively as a JSON streaming payload, it is called with the use of the `neuron-monitor` command. Unlike `nvidia-smi`, which is polled at each sampling interval with the inline call

```
nvidia-smi \
--query-gpu=utilization.gpu,memory.used,memory.total,
temperature.gpu \
--format=csv,noheader,nounits
```

retrieving the gpu utilization, memory utilization, memory total and gpu temperature metrics in a formatted CSV line, stored together with the Prometheus metrics in a temporary file, the `neuron-monitor` is initialized and terminated at each benchmark iteration and its telemetry data is retrieved in the temporary log file `/tmp/neuron_monitor.log`. Once GuideLLM completes a sweep, its output file is saved into the benchmark specific subfolder, alongside a CSV file created through the `process_metrics.py` script, written in the Appendix. The python code executes at each benchmark run and processes into a single structured file the Prometheus and the `nvidia-smi` data points or aggregating the flattened `neuron-monitor` logs. Finally, the `metrics.csv` file is converted into a `.parquet` format to reduce storage occupation and optimize query times for the analyses.

3.6. Dataset Description

Through the data collection and processing pipeline, the series of benchmarks produced two types of `metrics.parquet` datasets, one for each distinct hardware architecture explored. The columns that represent the common portion of these datasets include the sample's timestamp in epoch time format, the explicitly set vLLM inference hyperparameters, namely `timestamp`, `dtype`, `gpu_memory_utilization`, `max_num_seqs`, `max_num_batched_tokens` and the metrics extracted from the vLLM Prometheus endpoint, comprising:

- **Four built in `python` and six `process` metrics**: Included by default by `prometheus_client`, monitor Python garbage collection, information, process memory and CPU utilization. However, these are omitted from the analysis as they do not provide an aggregated view across the multi-process architecture that constitutes a vLLM engine instance [49].
- **85 `vllm:*` metrics**: Span the entire inference process and provide gauges, counters and timeseries at different aggregations. They include, for example, `vllm:time_to_first_token_seconds_sum`, `vllm:time_to_first_token_seconds_count`, `vllm:request_time_per_output_token_seconds_sum`, `vllm:request_time_per_output_token_seconds_count` to calculate TTFT and TPOT, and `vllm:kv_cache_usage_perc` for KV cache usage.
- **16 `http_*` metrics**: Calculate generic latency and throughput of HTTP requests.

All these are used to calculate the final metrics using simple arithmetic and the logic described in the Prometheus documentation for its query language *promql* [34]. The hardware-specific dataset columns include several percentiles of neuron_monitored metrics for the Inferentia2 chip, and the queried `nvidia-smi` metrics for the GPU instance. To generate a final aggregated dataset on which the analyses are based, the python code developed in the jupyter notebook — found in the Appendix — reads each subfolder content, parses and aggregates the main metric data points, generalizes the hardware accelerator-specific metrics and creates a dataframe of the following structure, one row per each of the 198 runs:

- `hardware`: set to "GPU (g4dn)" or "Neuron (inf2)", based on the instance on which the benchmark was executed.

- `model_family`: "Qwen" or "Llama", respectively referring to runs with model Qwen/Qwen2.5-3B-Instruct and meta-llama/Llama-3.2-1B.
- `experiment_group`: the composition of `hardware` and `model_family`.
- `max_num_seqs`, `max_num_batched_tokens`, `dtype`, `gpu_memory_utilization_key`: the values for the respective hyperparameters.
- `duration_sec`: the duration of the entire benchmark, calculated as the difference between the last and the first timestamp in the benchmark's dataset.
- `total_tokens`: the maximum value of metric `vllm:generation_tokens_total`, as the total tokens generated in the run.
- `throughput_tps`: the average throughput, result of `total_tokens / duration if duration > 0 else 0`.
- `avg_ttft_sec`: the average Time To First Token in seconds.
- `avg_tpot_ms`: the average Time Per Output Token (or Inter Token Latency) in milliseconds.
- `max_kv_cache_perc`: the maximum KV cache occupation percentage.
- `cost_per_1m_tokens`: the cost per million tokens, calculated as the ratio between the instance price rate and the number of tokens produced during the benchmark.
- `status`: indicates whether the run was successful or failed and the cause of failure.

4 | Empirical Performance and System Boundaries

Having established a deterministic benchmarking methodology to evaluate large language model serving across diverse hyperparameter configurations and two distinct inference architectures, and having executed a unified data aggregation pipeline, this chapter presents the resulting empirical performance data. The analyses include the pure performance comparison between the general-purpose machine learning acceleration of NVIDIA's GPU and the domain-specific architecture of AWS's Inferentia2 ASIC, as well as investigations on the impact of tuning inference hyperparameters on response time and throughput. The results are presented as graphs produced with the use of a Python notebook. The first analyses summarize the benchmark statistics, categorizing the runs in the different success or failure states and possible reasons. This chapter proceeds with several considerations on data representing the performance response to the hyperparameter tuning and the comparison between the two hardware architectures. The final section gathers the previous information and translates the performance-related analyses into the financial results, as the main objective of this thesis, by considering inference costs.

4.1. Run Reliability and System Failures

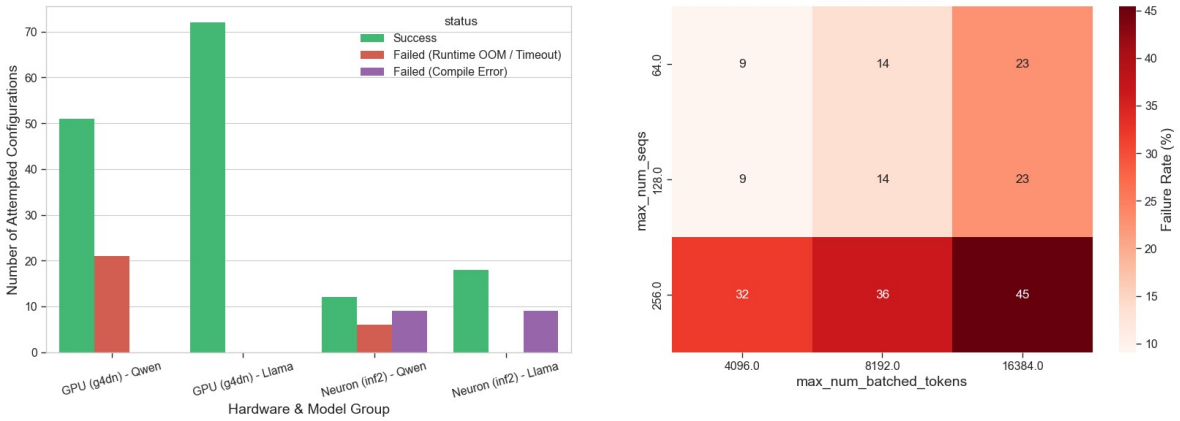
Of the 198 combinations of hyperparameters, models and chips, 153 produced a successful benchmark run; the remaining 45 either failed to complete the synthetic load test or errored during vLLM startup or pre-compilation. Broken down by experiment group (Figure 4.1a), the runtime-error failures occurred only with the 3-billion-parameter Qwen model: being the larger of the two models, it requires careful configuration to fit safely within the limited VRAM.

On the `inf2.8xlarge`, vLLM interfaces with the Ahead-of-Time (AOT) graph compilation model of the AWS Neuron SDK. Because the fixed KV cache size is proportional to `max_num_seqs`, during pre-compilation the model graph can exceed the physical host memory limit and cause an OOM. Empirically, all compile errors occurred during graph

building with 256 set as value for the concurrency limit.

Where compiled successfully, each hyperparameter configuration on Llama completed the benchmark, demonstrating a substantial operational margin and suggesting that the hardware is built to largely handle small model sizes with the considered settings.

The 22.7% total failure rate is decomposed by configured prefill buffer size `max_num_batched_tokens` and concurrency limit `max_num_seqs` in Figure 4.1b. As expected, the failure rate plot shows a worsening in the metric as the two hyperparameters are increased, proving how aggressive configurations expand the prefill and decoding memory footprint beyond available limits. On the GPU, an OOM error manifests as a standard `torch.OutOfMemoryError: CUDA out of memory` exception as vLLM attempts to preallocate the KV cache based on the `gpu_memory_utilization` configuration and the CUDA runtime crashes when its value is exceeded. The same error, when experienced on Inferentia2 architectures, causes the AWS Neuron Runtime to throw a memory allocation failure when handling peaks of allocation on the dedicated chip memory.



(a) Execution stability across cohort groups.

(b) Failure distribution by stress boundaries.

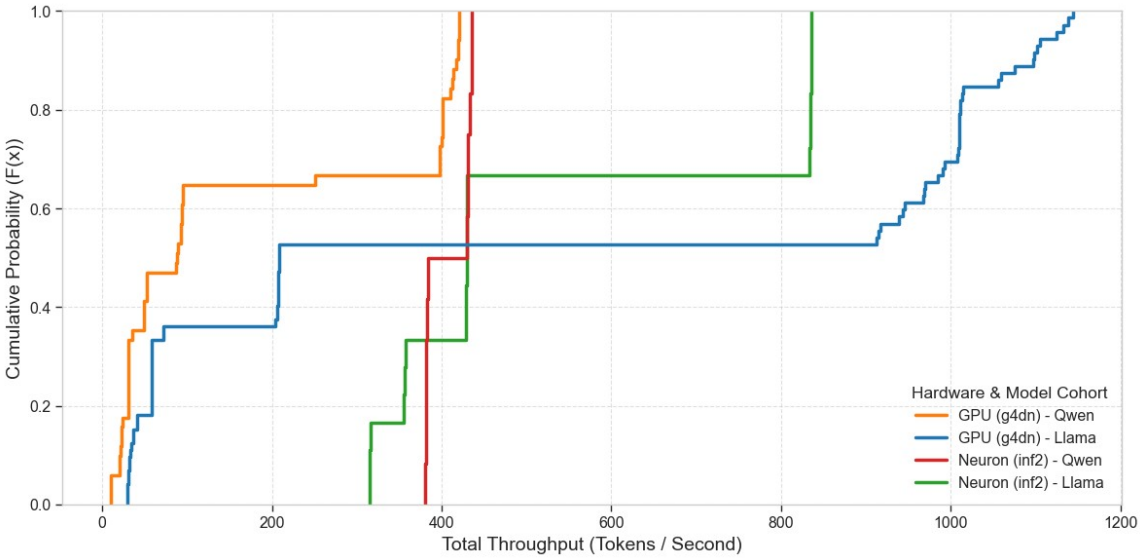
Figure 4.1: The images show a macro overview of execution stability across different cohort experimental groups alongside the failure distribution under progressive hyperparameter stress parameters.

As detailed in Section 4.4, it should be noted that raising the precision `dtype` from `float16` or `bfloat16` to `float32` contributes to memory allocation as the footprint of model weights and KV cache elements is doubled, getting closer to the threshold at which the system hits an OOM error.

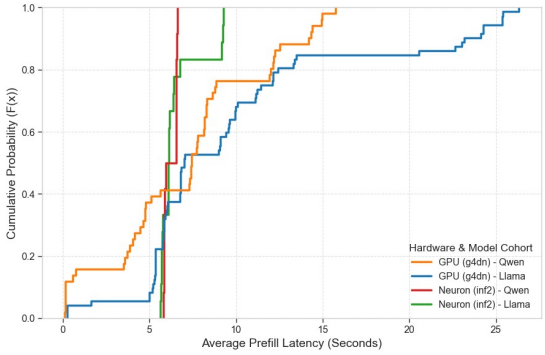
4.2. The Efficient Frontier

When analyzing pure performance of each benchmark execution, there is not a clear separation between the experimental categories by which an absolute best combination of hardware and model can be identified. The graphs in Figure 4.2 show the cumulative distribution function of throughput, time to first token and inter-token latency across all 153 successful benchmarks, color coded for each `experiment_group`.

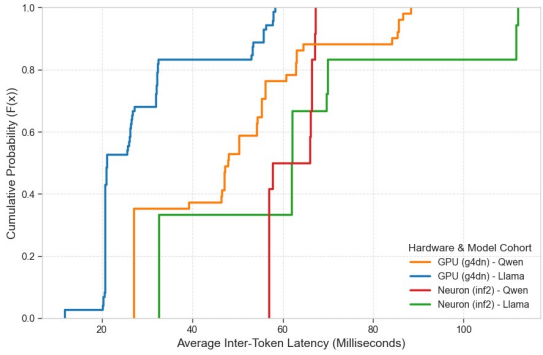
For throughput and ITL, the highest ranking model-hardware combination is `meta-llama/Llama-3.2-1B` on GPU. However, the same combination provided the highest prefill latency, which could be explained by slower request queuing caused by the increased throughput. The performance variability of the Llama model is to be attributed to the higher number of related successful benchmarks thanks its smaller, more manageable size, as the runs with other models failed more frequently under high-overhead configurations. The lowest TTFT was detected for Qwen/Qwen2.5-3B-Instruct, again on GPU, which is counterintuitive being this the larger model.



(a) Total system token throughput profile.



(b) Prefill phase latency ($TTFT$).



(c) Autoregressive decoding loop latency ($TPOT$).

Figure 4.2: The top chart displays the Empirical Cumulative Distribution Function (CDF) of processing throughput capacity across all successful runs. The bottom side-by-side plots map out the corresponding cumulative probability distribution for execution latency metrics across both the prompt prefill and token decoding generation phases.

The efficient frontier generated by throughput and latency, for each model, shows how the configurations sitting along or directly defining the curve that covers the lower rightmost part of the graph represent the optimal efficiency: for a specific latency budget, the points on the curve determine the best throughput achieved, or conversely, the minimum achievable latency for a given target throughput. The curves, which are reported in the graph in Figure 4.3 as dashed lines and highlighted points, are mainly defined by the GPU runs, with the Inf2 only defining the frontier for the highest throughput when running Qwen. The points scattered above or to the left of this line represent sub-optimal

configurations where the engine wastes hardware capacity or suffers from queue delays. The graph for Llama further proves the highly sensitive configuration space having the points widely dispersed across the canvas. The Qwen graph shows more tight, vertically stacked clusters, indicating a sensitivity for minor adjustments in batching or sequence limits, affecting TPOT without yielding additional throughput.

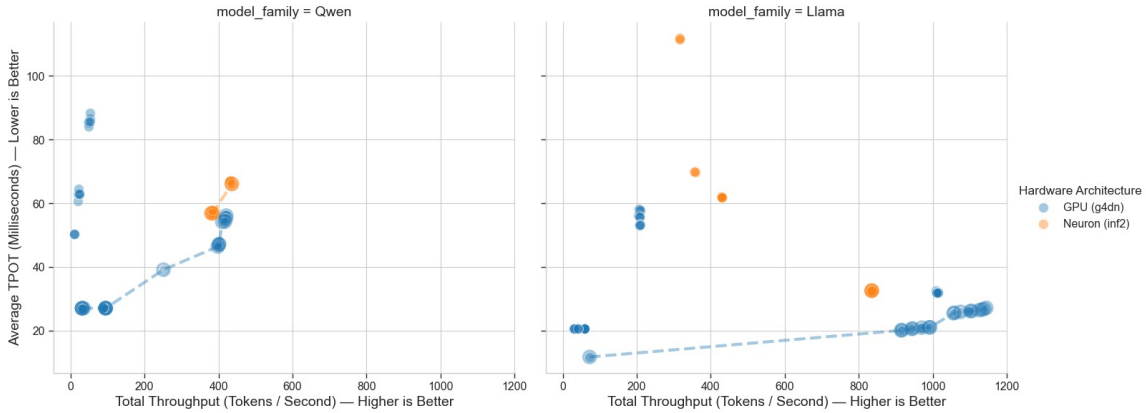


Figure 4.3: The scatter plots show the design space exploration of throughput against per-token latency ($TPOT$). The superimposed dashed curves delineate the empirical efficient frontier.

The optimal configuration for production serving is located at the "knee" of the Pareto line, the point where the curve shows a sharp turn and transitions from moving horizontally to spiking vertically. For Qwen, this point is located at around 400 tokens per second, pushing the engine past this knee represents an unstable or inefficient scenario. The knee for Llama is not clear since the curve maintains a stable trend at high throughput, the point could be found by applying increasingly resource-intensive configurations beyond the experimental grid.

4.3. Hyperparameter Interaction

The main impact of hyperparameter tuning was given by increasing concurrency, which correlates with higher KV cache consumption and longer prefill latency. Conversely, this parameter made a modest contribution in lowering TPOT. This behavior is interesting and unusual since the expected outcome of increasing `max_num_seqs`, by relaxing restriction on the number of requests processed simultaneously, is the exact opposite: the improvement of throughput and the degradation of inter-token latency.

The heatmaps in Figure 4.4, by displaying the average metric values for throughput and

latency, confirm the unexpected observed trend. The figure also shows the low variability in performance by tuning the maximum batched token count.

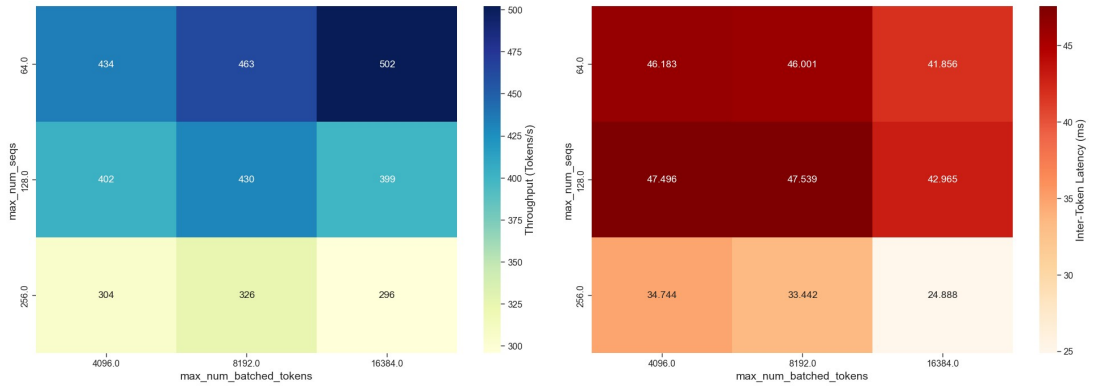


Figure 4.4: Average serving performance heatmaps. Throughput and inter-token latency (TPOT) variations across concurrency limits and batched token configurations.

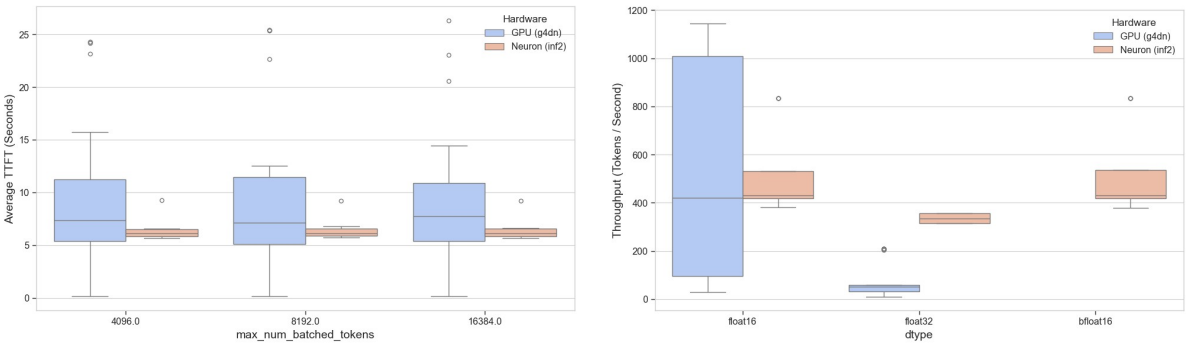
The described throughput paradox has been hypothesized to be caused by an aggregation artifact introduced during the dynamic rate-sweeping lifecycle. Because the benchmark evaluates run performance across a sequential concurrency sweep, the metrics reflect a global unweighted arithmetic mean over the entire duration of the trial. At elevated concurrency bounds, requests spend prolonged initial intervals in vLLM's waiting queue or executing chunked prefills. These prolonged low-throughput windows register localized metrics close to zero, mathematically dragging down the final `throughput_tokens_sec` captured in the dataset.

An alternative explanation could derive from the phenomena of *memory thrashing* [10] and *batch fragmentation* [26]: when setting `max_num_seqs` to high values, the system simultaneously tries to allocate an excessive amount of concurrent requests, saturating the accelerator's memory and consequently causing a throughput degradation by preempting requests and recomputing them, wasting numerous compute cycles. Because the scheduler constantly releases requests due to thrashing, the actual number of sequences actively being decoded drops. This fragmentation of the operational batch size results in the management of fewer concurrent requests, which could explain the observed reduction in TPOT.

4.4. Hardware Comparison

This section analyzes the observed differences in model serving for the two considered hardware types. As a bridge between the previous section and the current, the recorded numbers representing the impact of `max_num_batched_tokens` on Time To First Token

and of `dtype` on throughput are provided in Figures 4.5a and 4.5b, both metrics are split by hardware type. The first graph re-confirms the invariant nature of TTFT and its negligible correlation with the parameter controlling token capacity per step. In this and the following box plots, the limited height of the Neuron boxes clearly shows the higher stability and predictability in performance of the AWS ASIC, a relevant upside to be considered when deploying production-level, high reliability architectures. Minimizing the probability of failure enables the system to operate robustly and to be tuned to higher demanding configurations with reduced risk of OOM errors compared to a highly variant GPU deployment.



(a) Prefill Latency ($TTFT$) stability profile.

(b) Token throughput capacity scaling across precision formats.

Figure 4.5: Comparative hardware profiles showcasing prompt prefill stability across scaling token capacities (left) and throughput sensitivity under varying precision formats (right).

The median figures of TTFT, between GPU and Neuron, are also similar, which determines limited advantage in choosing GPU acceleration rather than the ASIC, the latter being less prone to configuration inefficiencies.

While still showing higher variance, the impact of precision on throughput on GPU is significantly more evident with respect to Neuron. For the baseline GPU architecture, the data shows that doubling the precision from `float16` to `float32` creates a severe performance bottleneck. It drops the median token generation rate by 8x, which is an 87% reduction (falling from 420.50 tok/s down to just 52.17 tok/s). This drastic drop occurs because 32-bit precision doubles the memory footprint required for model weights and intermediate activations. This increased overhead strictly limits the remaining VRAM allocation available for the KV cache, forcing the serving engine to throttle concurrent batch sizes. On the other hand, using `float16` widens the throughput range to a span of 1114.17 tok/s—stretching from a low of 30.49 tok/s up to a peak of 1144.66 tok/s. In com-

parison, the highest throughput achieved under `float32` is strictly limited to 208.56 tok/s. On inf2.xlarge, on which the `bfloat16` type is available, no variation between the two 16-bit parameter representations was recorded, while `float32` yielded a marginal reduction in performance, being the AOT memory scheduling more stable and predictable than the dynamic paradigm of CUDA.

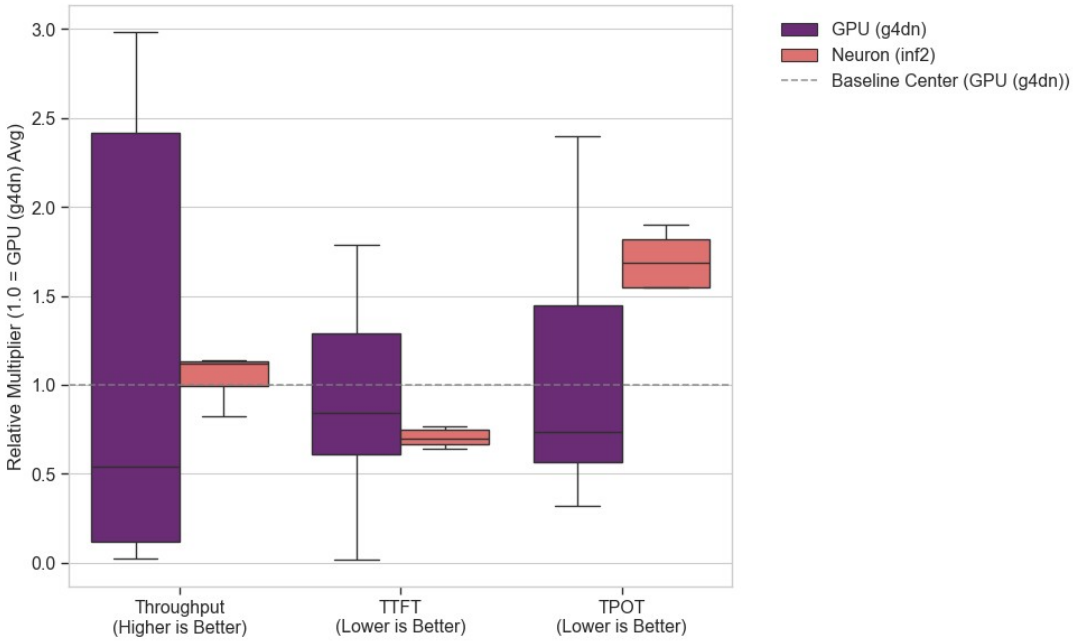


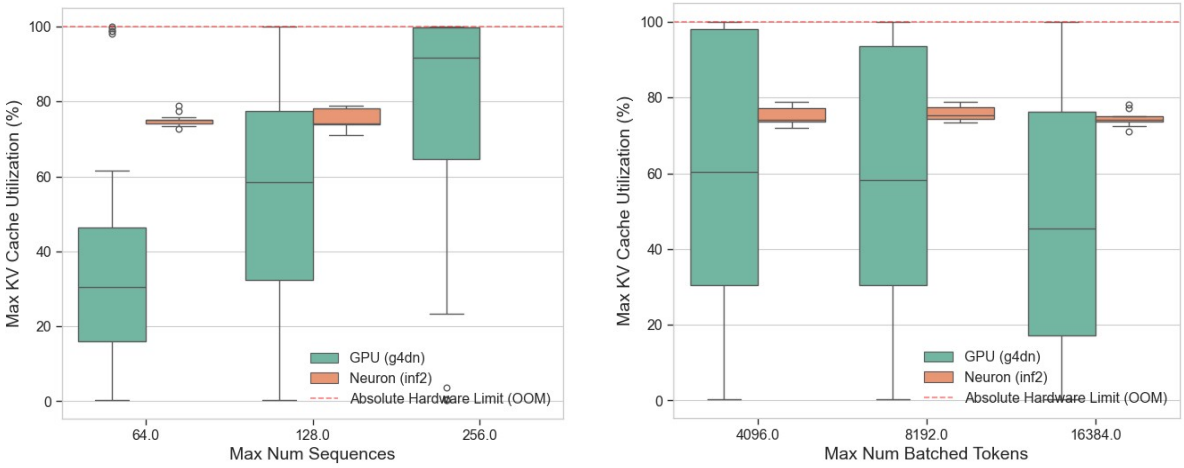
Figure 4.6: Hardware performance distributions normalized to the GPU baseline.

Between inf2.xlarge and g4dn.xlarge, the aggregated performance comparison, normalized on the baseline architecture's average performance in Figure 4.6, shows the stability of the Inferentia across the runs, compared to the GPU, which provided the optimal values for each metric, though being more volatile depending on hyperparameter configuration.

Analyzing the maximum KV Cache utilizations across the benchmark runs — split by the two memory allocation drivers in Figures 4.7a and 4.7b — it is proven that the dynamic allocation of the GPU is increasingly stressed under more sequence concurrency, while larger batches helped reduce the average utilization, making the GPU model serving less likely to fail for memory issues. This latter phenomenon represents a contradiction to what the `max_num_batched_tokens` hyperparameter should cause when increased, which is the growth of memory pressure, and is probably caused by the aggregation artifact or the artificial scheduler restrictions on the parallel active sequences explained in the previous section.

Conversely, the KV Cache pre-determined consumption on Neuron enabled the metric to be maintained in median higher w.r.t. GPU but stable between 70% and 80% through

each run.



(a) KV Cache utilization by concurrency limit.

(b) KV Cache utilization by number of batched tokens.

Figure 4.7: KV Cache saturation paths contrasting the elastic memory growth of dynamic CUDA frameworks against the rigid, deterministic footprints of static Neuron architectures.

4.5. Financial Efficiency Analysis

The main objective of this thesis is to identify the architecture and configuration that yields the highest performing LLM inference cloud deployment for a given running budget. The Pareto frontiers illustrated in Figures 4.8 provide the empirically optimal lines for maximizing a performance metric without increasing the cost, or, by transposing the relationship between the input and target space, minimizing the cost for a given performance SLA.

To translate the performance analysis of the previous sections into cost optimization insights, the additional raw variable to take into account is the price rate for the cloud instances on which the benchmarks were executed. The g4dn.xlarge EC2 machine type, which includes the NVIDIA T4 accelerator, is offered, at the moment of writing, at the hourly cost of 0.559 euros, while the inf2.xlarge, proposed as an ASIC-accelerated alternative for higher cost efficiency in machine learning inference, has a price of 0.967 euros per hour. For each experiment run, the cloud cost accumulated divided by the number of tokens produced, returns the cost per million output tokens. This metric is considered to be the fundamental measurement for the financial optimization of AI facilities, since the primary output in AI factories is intelligence manufactured in the

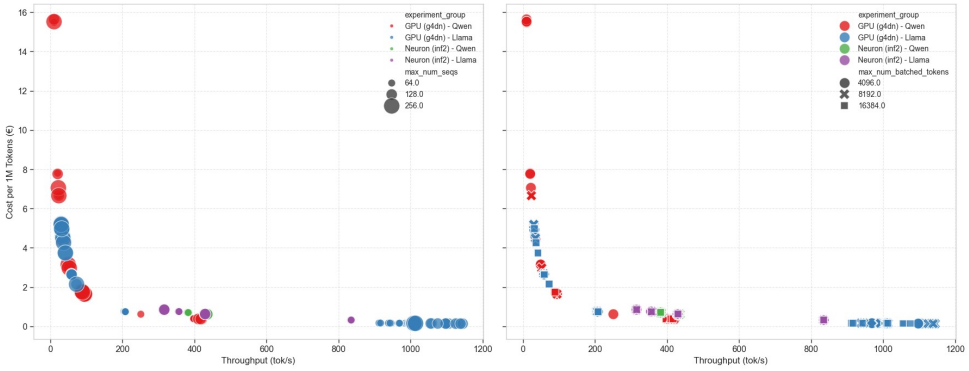
form of tokens.

Figures 4.8 show the financial frontier for each of the three performance metrics evaluated. In all graphs, the group of benchmarks executed on the Inferentia2 — represented by the points in green and purple — is positioned above the lower empirical boundary, which is almost entirely formed by the points related to the GPU runs. This positioning indicates a modest cost efficiency advantage of the GPU, yielding higher throughput, and lower inter-token and first-token latency with respect to the Neuron chip.

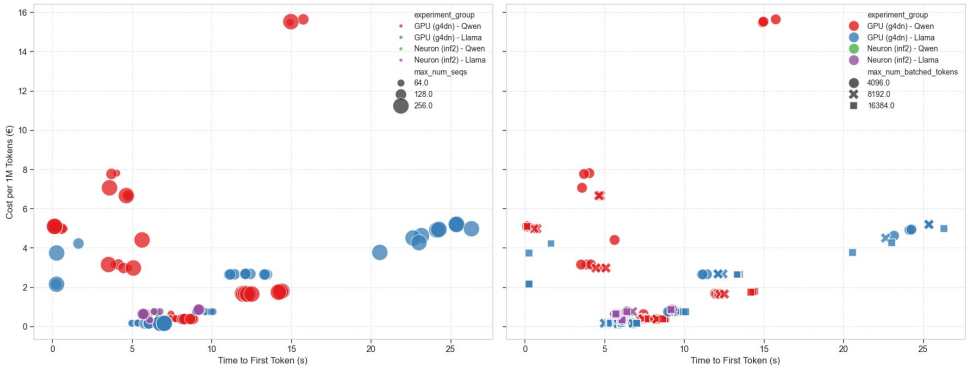
The lowest cost per million tokens has been recorded while running the models on GPU: €0.14 for Llama is the empirical lower bound. Moving on the axis representing the optimal price per token, reducing the maximum number of sequences yielded lower startup and inter-token latencies, while the highest throughput was produced with the same cost by setting `max_num_seqs` to 128 and `max_num_batched_tokens` to 8192. In batch processing production scenarios, where throughput is the main business driver, this configuration is the optimal among the ones considered in the experiment, as it provided the lowest cost and highest throughput with a peak of 1145 tokens per second for Llama. In contrast to the trend along the line of lowest cost, optimal latencies are reached with the highest values for concurrency and batched tokens. For applications where latency defines the quality of the deployment, the results highlight a large cost penalty as the lowest latency — 12 ms — is achieved by configuring `max_num_batched_tokens` to 16384 and `max_num_seqs` to 256, resulting in the prioritization of individual token generation speed over the throughput, which drops to 296 tps. The average performance and cost of the larger model were slightly worse than that of the smaller one. Compared to the cheaper model, Qwen led to a similar outcome in terms of cost optimization considerations and best performing configurations, proving the scalability of the analysis on larger and more expensive language models.

Ultimately, these frontier optimizations assume a rigid, static baseline dictated by standard On-Demand pricing. When evaluating the absolute lowest-cost configurations along the financial frontier, the baseline NVIDIA GPU architecture establishes a dominant, cost-effective entry point under On-Demand conditions, yielding a substantial ASIC-to-GPU cost ratio of 2.37x in favor of the GPU infrastructure.

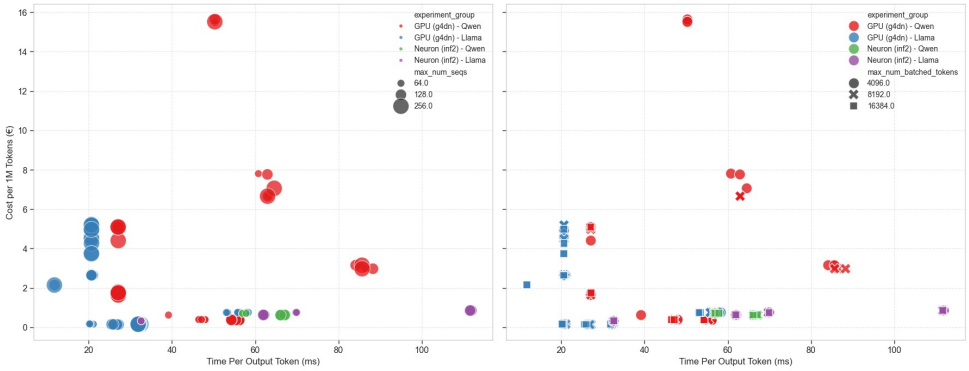
However, incorporating alternative billing structures — such as Spot Instance procurement — alters this frontier alignment by narrowing the competitive gap. Because the inf2.xlarge platform exhibits a massive Spot discount profile (a 6.36x reduction) compared to the g4dn.xlarge platform (a 2.79x reduction), transitioning to a spot model reduces the GPU's initial frontier advantage. Under Spot pricing, the ASIC's optimal frontier point plummets to a Cost Per Million Token (CPMT) of around €0.051, bringing



(a) Cost per 1M tokens versus inference throughput. The efficiency frontier converges in the bottom-right quadrant.



(b) Cost efficiency mapped against Time To First Token (TTFT). Optimal configurations cluster in the bottom-left quadrant.



(c) Cost efficiency relative to Time Per Output Token (TPOT). Optimal deployments in the bottom-left quadrant.

Figure 4.8: Financial efficiency frontiers mapping cost per million tokens to performance metrics, varying running environments and hyperparameters.

it within striking distance of the GPU's compressed boundary of around €0.049. This shifts the frontier cost ratio from 2.37x down to a nearly equal 1.04x.

Despite these dramatic cost reductions, leveraging Spot infrastructure introduces significant operational challenges that do not exist with On-Demand hosting. Because Spot instances utilize excess AWS capacity, they come with no availability guarantees; AWS can issue an eviction notice and reclaim the hardware with only a two-minute warning whenever market demand spikes. For large language model serving, this eviction behavior introduces severe engineering complexity. A sudden termination forces the system to gracefully handle interrupted inflight requests, manage user-state or context data losses mid-generation, and handle the lengthy cold-start overhead required to re-allocate new instances and reload large model weights into VRAM. While the baseline GPU maintains its financial edge at the frontier floor, this convergence highlights that the economic variance between architectures is deeply sensitive to the underlying cloud procurement model, forcing engineers to balance aggressive cost optimizations against system reliability.

Finally, it is critical to contextualize these findings within the respective hardware lifecycles of both platforms. The NVIDIA T4 GPU evaluated in this study is built on the older Turing architecture released in 2018, making it a significantly older piece of hardware compared to the AWS Inferentia2 ASIC, which was launched much later in 2023 specifically to optimize modern transformer workloads. Despite this five-year generational disadvantage, the older GPU infrastructure still managed to achieve, even if closely, the absolute lowest cost per token along the financial frontier across both billing models. Consequently, upgrading the baseline infrastructure to any newer generation of NVIDIA GPUs would only expand this performance gap, shifting the financial edge even further in favor of the GPU ecosystem.

5 | Conclusions and Future Developments

5.1. Summary of Contributions

This thesis aimed to observe and compare how different model serving architectures and configurations influenced the cost and performance of LLM inference deployments while maintaining the desired output quality and model behavior. The methodology for the experiments consisted of running several load generation and data collection pipelines on differently accelerated cloud machines, by varying the configuration of infrastructure-performance-related hyperparameters, to create an exhaustive dataset to be analyzed. The hardware accelerator faceoff was set as the benchmark between the gold standard GPU and the inference-optimized AWS Inferentia2 ASIC running on entry-level AWS EC2 virtual machines. The GPU determined the most efficient configurations on every optimization use case: throughput maximization, latency and cost minimization.

Serving on the custom ASIC of the inf2.xlarge instance, instead, proved to be the ideal solution for more performance stability, controllability and predictability as its precompiled architecture provided a stable and predetermined resource consumption, sitting slightly above the optimal cost optimization frontier.

5.2. Recommendations

As the translation of the empirical results into concrete recommendations, it is reasonable to state that, in application deployments where online inference needs to optimally exploit the hardware's compute power for highly variable loads, the engineering team should default to a dynamic GPU architecture. For predictable, high-volume offline batch processing workloads, instead, the Inferentia architecture is a safer option.

For hyperparameter tuning, varying the concurrency limit determined the most noticeable difference in model serving performance. The experiments highlighted a counterintuitive

degradation of throughput with the use of higher values for `max_num_seqs`. Therefore the strict bounding of such parameter is recommended to protect the KV cache and prevent system preemptions.

The final analysis, determining the empirical financial frontier, provided `max_num_seqs = 128, max_num_batched_tokens = 8192` as the configuration with the lowest cost. Tuning on these values also achieved the highest throughput and can represent a starting point to improve latency, navigating the Pareto frontier, by lowering `max_num_seqs`. Setting this parameter to 64 in the benchmark lowered TPOT to 47 milliseconds with a marginal increase in cost and drop in throughput. Due to the paradox identified and detailed in Chapter 4, the absolute minimum inter-token latency was achieved by raising concurrency to 256. Being the artificial byproduct of system instability, probably related to the specific experimental setting, this configuration is not recommended as optimal for latency reduction. Furthermore, it resulted in a significant drop in throughput and a substantial price increase, not justifiable by the improvement in latency.

5.3. Limitations of the Study

In this section are listed the limiting aspects to take into account when considering the results of this benchmark. First, when evaluating machine costs, the study focused solely on AWS On-Demand prices. For long-term production deployments, pricing models beyond standard hourly rates can significantly shift the economic evaluation. For example, Amazon EC2 Spot Instances offer steep discounts but introduce availability risks, as AWS can reclaim the capacity at any time. Interestingly, this model heavily favors the ASIC architecture; the inf2.xlarge spot price drops by 6.4x (from €0.967/h to €0.152/h), whereas the g4dn.xlarge spot price only decreases by 2.8x (from €0.559/h to €0.20/h). Alternatively, adopting 1-year or 3-year Reserved Instance plans alters the cost proportions between these platform types, which could make the ASIC more cost-effective than the GPU under specific configurations. More expensive architectures, beyond the architectural scale available for the experiments, permit various types of parallelism — *Data Parallelism, Pipeline Parallelism, Tensor Parallelism* — that enable the exploitation of multiple nodes and architectural cores to dynamically scale computation and provide fault tolerance. The loads considered are bounded by GuideLLM's deterministic traffic patterns. Real-world requests are volatile, contain unpredictable spikes and highly skewed prompts and output lengths, which are difficult to reproduce by load generators and affect the benchmarks' repeatability. Only one benchmark run per configuration has been executed. Allocating more time for the experiments to repeat the runs would help accounting

for performance variability and random fluctuations. Finally, the empirical conclusions of this study are strictly derived from dense, decoder-only Transformer models within the 1B to 3B parameter range utilizing Grouped-Query Attention. Evaluating alternative paradigms, such as Mixture of Experts networks, frontier models exceeding tens of billions of parameters, or architectures relying on different mathematical operators, would fundamentally alter the system's compute-to-memory ratio, potentially shifting the observed efficiency frontiers and optimal hyperparameter configurations.

5.4. Future Work

The possible future developments enabled by this thesis include the expansion of the empirical grid of architectures, hyperparameter values, and model sizes to achieve greater result generality. Secondly, the dataset collected in these experiments can be further studied by considering the metrics and aggregations captured but not included in this thesis' analysis. Furthermore, the datasets can be used to train a predictive optimization model. This would allow engineers to input specific Service Level Agreements and workload geometry (such as expected prompt lengths) to generate the mathematically optimal hardware choice and hyperparameter configuration, removing the need for costly empirical testing. Future research could implement a more realistic synthetic traffic generation rather than fixed input and output lengths, originally set in these experiments to avoid performance issues. Along with the repetition of performance runs, this would make the empirical dataset more robust and sound in representing real-life traffic variability. For completeness of optimization, the research should also broaden the scope of the analysis to account for model response quality. By instrumenting LLM observability frameworks, researchers could investigate whether aggressive hyperparameter tuning and continuous batching negatively impact the semantic quality or reasoning capabilities of the generated text. Finally, combining these empirical baselines with real-time observability could provide a foundation for a cost-aware dynamic request routing mechanism. Such system could dynamically route latency-sensitive requests to GPU instances while offloading high-throughput background tasks to ASIC architectures, optimizing the financial efficiency frontier continuously.

Bibliography

[1] U. A. vllm vs text generation inference: Llm serving benchmarks that actually matter. *Build MVP Fast*, 2026. URL https://www.buildmvpfast.com/blog/vllm-vs-tgi-llm-serving-benchmarks-2026.

[2] Y. AI. Best llm inference engines in 2026: vllm, tensorrt-llm, tgi, and sglang compared. *Yotta AI*, 2026. URL https://www.yottalabs.ai/post/best-llm-inference-engines-in-2026-vllm-tensorrt-llm-tgi-and-sglang-compared.

[3] AI@Meta. The llama 3 herd of models. arXiv preprint arXiv:2407.21783, 2024. URL https://arxiv.org/abs/2407.21783.

[4] Amazon Web Services. *Model Architecture Fit Guidelines for AWS Neuron*, 2024. URL https://awsdocs-neuron.readthedocs-hosted.com/en/latest/general/arch/model-architecture-fit.html. Official guidance on which LLM architectures are optimized for Inferentia2 hardware.

[5] Amazon Web Services. Running Inference on Variable Input Shapes with Bucketing. Application note, Amazon Web Services, 2026. URL https://awsdocs-neuron.readthedocs-hosted.com/en/latest/about-neuron/appnotes/torch-neuron/bucketing-app-note.html.

[6] Amazon Web Services. Generative LLM Inference with Neuron and KV-Cache Optimization. Application note, Amazon Web Services, 2026. URL https://awsdocs-neuron.readthedocs-hosted.com/en/latest/general/appnotes/transformers-neuronx/generative-llm-inference-with-neuron.html.

[7] Amazon Web Services. *Neuron DLAMI User Guide*. Amazon Web Services, Inc., 2026. URL https://awsdocs-neuron.readthedocs-hosted.com/en/latest/dlami/index.html.

[8] Amazon Web Services. Trainium/Inferentia2 Architecture Guide for NKI. Technical documentation, Amazon Web Services, 2026. URL https://awsdocs-neuron.readthedocs-hosted.com/en/latest/nki/guides/architecture/trainium_inferentia2_arch.html.

[9] S. Analysts. Nvidia ai gpu market share 2024-2026: 87 Technical report, Silicon Analysts, 2 2026. URL https://siliconanalysts.com/analysis/nvidia-ai-accelerator-market-share-2024-2026.

[10] M. Arif, A. Maurya, S. S. Vazhkudai, and B. Nicolae. Understanding inference scaling for llms: Bottlenecks, trade-offs, and performance principles, 2026. URL https://arxiv.org/abs/2605.19775.

[11] AWS, 2026. URL https://docs.aws.amazon.com/dlami/latest/devguide/what-is-dlami.html.

[12] Benchmarking, 2026. URL https://dictionary.cambridge.org/dictionary/english/benchmarking.

[13] G. C. Blog. Our eighth generation tpus: two chips for the agentic era. *Google Cloud Infrastructure & AI*, 2026. URL https://blog.google/innovation-and-ai/infrastructure-and-cloud/google-cloud/eighth-generation-tpu-agentic-era.

[14] T. Counterpoint. Ai server compute asic shipments to triple by 2027 as custom silicon enters hyper-growth phase. *Counterpoint*, 2026. URL https://counterpointresearch.com/en/insights/AI-Server-Compute-ASIC-Shipments-to-Triple-by-2027.

[15] S. Dubey, J. Park, S. S. Kharbanda, RJ, R. Goutam, S. Agarwal, and Y. Zhou. How rufus doubled their inference speed and handled prime day traffic with aws ai chips and parallel decoding. *AWS Blog*, 2025. URL https://aws.amazon.com/it/blogs/machine-learning/how-rufus-doubled-their-inference-speed-and-handled-prime-day-traffic-with-aws-ai-c

[16] A. Editorial. Nvidia's cuda lock-in and supply scarcity make its ai chip moat harder to break than it looks. *AlphaStreet News*, 2026. URL https://news.alphastreet.com/nvidias-cuda-lock-in-and-supply-scarcity-make-its-ai-chip-moat-harder-to-break-thar

[17] A. Goel and R. Borgohain. vllm vs. tgi: The ultimate comparison for speed, scalability, and llm performance. *Inferless*, 2024. URL https://www.inferless.com/learn/vllm-vs-tgi-the-ultimate-comparison-for-speed-scalability-and-llm-performance.

[18] Google, 2026. URL https://cloud.google.com/tpu.

[19] G. D. C. Hub. 19 key takeaways from jensen huang's nvidia gtc 2026 keynote. *Global*

Data Center Hub Reports, 2026. URL https://www.globaldatacenterhub.com/p/19-key-takeaways-from-jensen-huangs-f2c.

[20] HuggingFace. Aws trainium & inferentia documentation: Supported architectures, 2026. URL https://huggingface.co/docs/optimum-neuron/supported_architectures.

[21] HuggingFace, 2026. URL https://huggingface.co/docs/text-generation-inference/index.

[22] Instances, 2026. URL https://instances.vantage.sh/aws/ec2/g4dn.xlarge?currency=EUR&region=eu-central-1.

[23] Instances, 2026. URL https://instances.vantage.sh/aws/ec2/inf2.xlarge?currency=EUR&region=eu-central-1.

[24] Intel, 2026. URL https://www.intel.com/content/www/us/en/learn/fpga-for-ai.html.

[25] D. Kalamkar, D. Mudigere, N. Mellempudi, D. Das, K. Banerjee, S. Avancha, B. Kaul, E. Georganas, A. Heinecke, P. Pradhan, et al. A Study of BFLOAT16 Floating-Point Format for Deep Learning Training and Inference. *arXiv preprint arXiv:1905.12322*, 2019. URL https://arxiv.org/abs/1905.12322.

[26] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, New York, NY, USA, 2023. ACM, Association for Computing Machinery. URL https://arxiv.org/abs/2309.06180.

[27] V. Liagkou, E. Filiopoulou, G. Fragiadakis, M. Nikolaidou, and C. Michalakelis. The cost perspective of adopting large language model-as-a-service. *2024 IEEE International Conference on Joint Cloud Computing (JCC)*, 2024. URL https://www.researchgate.net/publication/384377213_The_cost_perspective_of_adopting_Large_Language_Model-as-a-Service.

[28] R. Liang. Scaling ai without breaking the grid: The path to sustainable innovation. *World Economic Forum*, 2025. URL https://www.weforum.org/stories/2025/01/scaling-ai-without-breaking-the-grid.

[29] R. Liang. How to reduce your cloud costs for ai. *Crusoe*, 2025. URL https://www.crusoe.ai/resources/blog/how-to-reduce-your-cloud-costs-for-ai.

[30] I. Neural Magic. Guidellm: Scalable inference and optimization for large language models. https://github.com/vllm-project/guidellm, 2024.

[31] NVIDIA Corporation. CUDA Programming Guide. Technical documentation, NVIDIA, 2026. URL https://docs.nvidia.com/cuda/cuda-programming-guide/.

[32] NVIDIA Corporation. *NVIDIA System Management Interface (nvidia-smi) Command Line Utility*, 2026.

[33] R. Pope, S. Douglas, A. Chowdhery, J. Devlin, J. Bradbury, J. Heek, K. Xiao, S. Agrawal, and J. Dean. Efficiently scaling transformer inference. arXiv preprint arXiv:2211.05102, 11 2022. URL https://arxiv.org/abs/2211.05102.

[34] Prometheus Authors. *Querying Basics*. The Prometheus Project, 2026. URL https://prometheus.io/docs/prometheus/latest/querying/basics/.

[35] Revendra. Powering the ai revolution: A business leader's guide to the gpu and ai accelerator market. *LinkedIn Business Insights*, 2025. URL https://www.linkedin.com/pulse/powering-ai-revolution-business-leaders-guide-gpu-market-t-ph-d-xxxtc.

[36] J. Schneider and I. Smalley. What is a neural processing unit (npu)? *IBM Think*, 2024. URL https://www.ibm.com/think/topics/neural-processing-unit.

[37] R. Sennrich, B. Haddow, and A. Birch. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, pages 1715–1725. Association for Computational Linguistics, 2016. URL https://aclanthology.org/P16-1162/.

[38] A. W. Services, 2026. URL https://aws.amazon.com/it/ec2/pricing/on-demand/.

[39] A. Shimes. Why hardware prices are rising in 2025–2026 | it supply chain trends. *Telefonica Tech*, 2026. URL https://telefonicatech.uk/articles/why-hardware-prices-are-rising-in-2025-2026/.

[40] J. Su, M. Ahmed, Y. Lu, S. Pan, W. Bo, and Y. Liu. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024. URL https://arxiv.org/abs/2104.09864.

[41] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural

networks. In *Advances in Neural Information Processing Systems*, volume 27, pages 3104–3112, 2014. URL https://arxiv.org/abs/1409.3215.

[42] H. Team. Tpus vs gpus vs asics: Ai hardware guide 2025. *HowAIWorks.ai*, 2025. URL https://howaiworks.ai/blog/tpu-gpu-asic-ai-hardware-market-2025.

[43] M. Temkin. Inference startup inferact lands $150m to commercialize vllm. *tech Crunch*, 2026. URL https://techcrunch.com/2026/01/22/inference-startup-inferact-lands-150m-to-commercialize-vllm/.

[44] H. Umesh. Ollama vs. vllm: A deep dive into performance benchmarking. *Red Hat Developer*, 2025. URL https://developers.redhat.com/articles/2025/08/08/ollama-vs-vllm-deep-dive-performance-benchmarking#.

[45] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, pages 5998–6008. Curran Associates, Inc., 2017. URL https://arxiv.org/abs/1706.03762.

[46] R. Verdecchia, J. Sallou, and L. Cruz. A systematic review of green ai. *Wiley Interdisciplinary Reviews*, 2023. URL https://robertoverdecchia.github.io/papers/WIRE_2023.pdf.

[47] vLLM, 2026. URL https://vllm.ai/.

[48] vLLM Project Team. vLLM Engine Arguments. https://docs.vllm.ai/en/stable/configuration/engine_args/, 2026.

[49] vLLM Project Team. vLLM Metrics. https://docs.vllm.ai/en/stable/design/metrics/, 2026. Accessed: 2026-05-17.

[50] vLLM Team. Integration with Hugging Face: vLLM Design Documents. Technical documentation, vLLM Project, 2026. URL https://docs.vllm.ai/en/latest/design/huggingface_integration/.

[51] A. Yang, B. Yang, B. Hui, B. Zheng, B. Yu, C. Zhou, C. Li, C. Li, D. Liu, F. Huang, et al. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*, 2024. URL https://arxiv.org/abs/2407.10671.

[52] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538,

Carlsbad, CA, USA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/yu.

A | Appendix: benchmark execution scripts

This appendix contains the scripts used to execute the benchmarks, collect the datasets and perform the final analysis.

All scripts are uploaded in the GitHub repository at the following URL: `https://github.com/anesaNico/benchmarking_thesis_repository`

`run_experiments_neuron.sh` is the bash script used to run the tests on the inf2.xlarge (with the Inferentia2 ASIC) instance, minor changes and simplifications are needed to execute with the same logic on GPU.

The python code in `process_metrics.py` is the auxiliary script called by the previous to process the collected metrics into one `.parquet` dataset.

`neuron_compile_models.py` is the python code used to trigger the compilation of the models on inf2.8xlarge. This models would then be exported and loaded on the inf2.xlarge to conduct the benchmarks.

`analysis.ipynb` represents the content of the Jupyter notebook used to analyze the final dataset and plot the results.

List of Figures

2.1 vLLM vs. Ollama benchmark 13

4.1 Empirical Evaluation of System Reliability and Collapse Boundaries 42
4.2 Empirical Cumulative Distribution Functions of Performance Profiles . . . 44
4.3 Empirical Efficient Frontier and Pareto-Optimal Boundaries 45
4.4 Average serving performance heatmaps. Throughput and inter-token latency (TPOT) variations across concurrency limits and batched token configurations. 46
4.5 Comparative hardware profiles showcasing prompt prefill stability across scaling token capacities (left) and throughput sensitivity under varying precision formats (right). 47
4.6 Hardware performance distributions normalized to the GPU baseline. . . . 48
4.7 KV Cache saturation paths contrasting the elastic memory growth of dynamic CUDA frameworks against the rigid, deterministic footprints of static Neuron architectures. 49
4.8 Financial efficiency frontiers mapping cost per million tokens to performance metrics, varying running environments and hyperparameters. 51

List of Tables

2.1 LLM serving throughput benchmarks comparing vLLM and TGI across different model sizes. 14
2.2 Latency percentile benchmarks comparing vLLM and TGI, highlighting the trade-offs between Time to First Token (TTFT) and end-to-end processing intervals. 14
2.3 Summary of architectural and execution divergences between GPUs and AWS Inferentia2 during LLM serving. 18

3.1 Specifications for EC2 G4dn instances featuring NVIDIA T4 GPUs. Source: `https://aws.amazon.com/it/ec2/instance-types/g4` 25
3.2 Specifications for EC2 Inf2 instances powered by AWS Inferentia2 chips. Source: `https://aws.amazon.com/it/ec2/instance-types/inf2` 26
3.3 Hardware specifications summary for the evaluated AWS EC2 instances. . 27
3.4 Experimental hyperparameter search grid configuration showing execution scopes across both dynamic GPU and static ASIC hardware environments. 36